TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: N 2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie

# *Testování a optimalizace metody rozkladu oblasti pro specifické úlohy proudění*

# *Testing and optimization of domain decomposition method for particular flow problems*

Diplomová práce

Autor: Tomáš Bambuch
Vedoucí práce: Mgr. Jan Březina, Ph.D.

V Liberci 20. 5. 2011

TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky, informatiky a mezioborových studií

Akademický rok: 2010/2011

# ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

| | |
|---|---|
| Jméno a příjmení: | **Bc. Tomáš BAMBUCH** |
| Osobní číslo: | **M08000190** |
| Studijní program: | **N2612 Elektrotechnika a informatika** |
| Studijní obor: | **Informační technologie** |
| Název tématu: | **Testování a optimalizace metody rozkladu oblasti pro specifické úlohy proudění** |
| Zadávající katedra: | **Ústav nových technologií a aplikované informatiky** |

Zásady pro vypracování:

1. provést testy škálovatelnosti proudění (lineární solver) a transportu (maticové násobení) na akademických i reálných problémech, jednak na počítači se sdílenou pamětí a dále na ethernetovém clusteru
2. optimalizovat překlad programu i podpůrných knihoven na clusteru
3. optimalizovat nastavení některých numerických parametrů předpodmiňovače a solveru

Rozsah grafických prací: **dle potřeby**

Rozsah průvodní zprávy: **cca 50 stran**

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

**[1] SMITH, Barry; BJORSTAD, Petter; GROPP, William. Domain decomposition: parallel multilevel methods for elliptic partial differential equations. Cambridge: Cambridge University Press, 2004. 240s. ISBN 978-0521602860**

**[2] PETSc Manual [online]. URL: http://www.mcs.anl.gov/petsc/petsc-as**

Vedoucí diplomové práce: **Mgr. Jan Březina, Ph.D.**

Ústav nových technologií a aplikované informatiky

Datum zadání diplomové práce: **15. října 2010**

Termín odevzdání diplomové práce: **20. května 2011**

prof. Ing. Václav Kopecký, CSc.
děkan

prof. Dr. Ing. Jiří Maryška, CSc..
vedoucí ústavu

V Liberci dne 15. října 2010

# Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

Datum: 20. 5. 2011

Podpis:

# Poděkování

Na tomto místě bych chtěl poděkovat těm, kteří mi pomáhali při psaní této diplomové práce, především pak vedoucímu Mgr. Janu Březinovi, Ph.D. za jeho trpělivost, ochotu a za veškerý čas, po který se mi během konzultací věnoval. Mé díky patří i RNDr. Petru Pospíšilovi, CSc. za cenné rady a připomínky při testování na superpočítači Rex.

# Abstrakt

Tato práce se zabývá problematikou paralelních výpočtů, zejména pak s přihlédnutím k jedné oblasti jejich využití – simulaci proudění podzemní vody. První část práce se zaobírá základními veličinami, statistikami a zákony, se kterými se můžeme u dané problematiky setkat. Nechybí však ani popis softwarových nástrojů či rozhraní pro paralelní výpočty ani úvod do numerických knihoven, díky kterým můžeme požadované úlohy efektivně řešit. Věnujeme se zde i hardwaru, takzvaným superpočítačům, jejich architekturám a rovněž i popisu dvou počítačů, které byly použity při testování ve druhé části práce. Druhá část je již ryze praktická – základní porovnání obou testovaných počítačů poskytují provedené testy ukázkových úloh z knihoven PETSc, které jsou zde rozebrány. Dále se práce zaměřuje na program pro simulaci proudění podzemní vody a transportu látek v horninovém prostředí, Flow123d, a to včetně popisu provedených úprav – implementaci tříd pro měření časových intervalů a sadu skriptů pro automatizované testování úloh. Jsou zde rozebrány i vlastní testovací úlohy programu Flow123d provedené na obou testovacích počítačích, a to včetně zhodnocení jejich výsledků.

Klíčová slova: paralelní programování, cluster, Flow123d, PETSc

# Abstract

This thesis is concerned with problems of parallel computing, especially with regard to one area of its use – the simulation of underground water flow. The first part of the work is discussing the basic variables, statistics and laws which we may encounter in our field of interest. A short description of software tools and interfaces for parallel computing is also given, together with an introduction to numerical libraries, by which the desired tasks can be effectively solved. We also deal with hardware, the so-called supercomputers and their architectures as well as with a description of two computers, which were used during our tests described in the second part. The second part is purely practical – tests made on the sample tasks included in PETSc libraries are analyzed there and thus provide basic comparison of both tested computers. The work is also focusing on software for water flow, solute transport and sorption in a heterogonous porous and fractured medium called Flow123d and mentions the modifications made in the program noted above – implementation of classes for time measurement and a set of scripts for automated tasks testing. Testing sets for program Flow123d performed on both tested supercomputers are analyzed there too, including the evaluation of the results.



Keywords: parallel programming, cluster, Flow123d, PETSc

# Obsah

# 1 Úvod

Předpověď počasí, modelování či kryptoanalýza – to jsou jen některé z oblastí, ve kterých je zapotřebí enormní výpočetní výkon, jenž by byl klasickými stolními počítači nedosažitelný. Ke slovu tu přichází takzvané superpočítače – těmi obecně rozumíme počítače, jejichž výpočetní výkon je několikanásobně vyšší než výklon klasických stolních počítačů. Na nich se potřebný výpočet provádí současně na více procesorech, neboli paralelně. Paralelismus představuje efektivní cestu ke zvýšení výkonu, jelikož možnosti dnešních procesorů jsou, co se týče zvyšování frekvence, velmi omezené. Zjednodušeně můžeme říci, že většina aplikací na paralelním počítači daný problém rozdělí na několik menších úloh, které se na více procesorech řeší současně, a po skončení výpočtu se jednotlivé dílčí výsledky opět „seskládají“ dohromady v jeden globální výsledek.

Umění navrhovat, analyzovat a verifikovat kvalitní paralelní algoritmy je oproti témuž umění v sekvenčním programování o to těžší, že zde přibývá nový rozměr: počet procesorů a způsob, jakým spolu mohou komunikovat a spolupracovat. V posledních zhruba padesáti letech bylo vymyšleno mnoho více i méně zajímavých a praktických řešení na různých paralelních architekturách, které se vyvíjejí souběžně s vývojem algoritmů.

Jedním z projektů spoléhajících na paralelní zpracování a řešení úloh je i simulátor proudění vody a transportu látek v horninovém prostředí nazvaný Flow123d, který je vyvíjen na Fakultě mechatroniky Technické univerzity v Liberci. Je důležité, že program na paralelním počítači funguje a vrací korektní výsledky, hodnotné jsou však i informace o tom, jak se program na takovém paralelním systému chová. Naším úkolem tak v této práci bude, mimo jiné, zjistit závislosti mezi dobou běhu programu pro různé úlohy a různé počty procesorů, případně identifikovat vzniklé anomálie a pokud možno určit jejich příčiny a v neposlední řadě také vzájemně porovnat dva superpočítače, každý s poněkud odlišnou architekturou.

# 2 Paralelní výpočty

Při vyhodnocování efektivnosti algoritmů si obvykle klademe otázku: Jestliže velikost problému (tj. velikost či počet vstupních dat) poroste, jak se algoritmus bude chovat, jak poroste jeho čas běhu? Teorie složitosti se obvykle zabývá asymptotickými tendencemi funkcí popisujících složitosti. My se zde nebudeme do detailu zaobírat teorií složitosti a funkcemi popisujícími tuto složitost, ale bude nás spíše zajímat ta „praktičtější“ stránka, kterou využijeme při našich testech, uvedených v kapitolách 5 a 7. Nejen v těchto kapitolách se však vyskytují některé pojmy, které nemusí být obecně známé a které si proto krátce představíme v následujících několika odstavcích.

## 2.1 Amdahlův zákon

V souvislosti s paralelními výpočty se často objevuje pojem takzvaného Amdahlova zákona. Tento zákon pochází z počáteční éry paralelních počítačů a popisuje fakt, že i při použití velkého množství procesorů lze obvykle dosáhnout jen omezeného zrychlení. Základem je předpoklad, že daný algoritmus obsahuje nějakou složku *P*, která je (nekonečně) paralelizovatelná, a oproti tomu zbylá část algoritmu, kterou označíme *1-P*, bude vždy prováděna sekvenčně. Výsledné zrychlení pro *N* procesorů by se dalo zapsat jako

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

Z čehož je patrné, že i při obrovském stupni paralelizace (N -> ∞) nikdy neeliminujeme onu sekvenční složku programu, zrychlení nemůže být nikdy vyšší než *1/(1-P)*. Pokud tedy budeme řešit problém, kde sekvenční část bude tvořit 10%, dosáhneme maximálně desetinásobného zrychlení. Nebude tedy prakticky záležet na tom, bude-li problém řešit 32 procesorů nebo například 1024.

## 2.2 Gustafsonův zákon

Na rozdíl od Amdahlova zákona, který předpokládá, že chceme problém fixní velikosti vyřešit v co nejkratším čase, Gustafsonův zákon uvažuje mnohem praktičtější scénář, a to ten, kdy s rostoucí paralelizací nepotřebujeme snižovat dobu výpočtu, ale prioritní je získat přesnější výsledek či zpracovat více dat. Místo problému konstantní velikosti zde tedy uvažujeme konstantní dobu běhu. Zrychlení potom můžeme zapsat ve tvaru

$$S = a(n) + N \cdot (1 - a(n))$$

Kde $N$ je opět počet procesorů a $a(n)$ představuje sekvenční část programu na paralelním stroji. Ta není s rostoucí velikostí problému konstantní, a proto ji zde máme vyjádřenou jako funkci, kde $n$ reprezentuje právě velikost řešené úlohy. V případě Amdahlova zákona jsme měli problém o konstantní velikosti, a tudíž jsme místo funkce sekvenční části mohli uvažovat pouze konstantu. U Gustafsonova zákona tkví myšlenka v tom, že s rostoucí velikostí problému se sekvenční složka obvykle příliš nemění, zatímco paralelní složka narůstá, a tudíž $a(n)$ jako funkce (poměr sekvenční složky v programu) postupně klesá. Podobně jako v případě Amdahlova zákona s rostoucím počtem procesorů roste i zrychlení, tentokrát ovšem není shora omezeno! Znamená to, že pro dostatečně velký problém můžeme dosáhnout libovolného zrychlení.

Jak je možné, že mohu docílit libovolného zrychlení, když ani jeden zákon neobsahuje žádný logický spor, avšak Amdahlův zákon tvrdí opak? Důvodem je skutečnost, že každý ze zákonů na problematiku nahlíží z jiného úhlu a pracuje s jinými proměnnými veličinami. V Amdahlově verzi se zabýváme možným zrychlením problému o konstantní velikosti a s konstantní sekvenční částí - v tomto případě je možné zrychlení skutečně omezeno. Gustafsonův zákon si za konstantu volí dobu běhu na paralelním systému a říká, že zrychlení není omezeno, ovšem za předpokladu dostatečně velkého problému.

## 2.3 Efektivita

Efektivitou se v teorii složitosti rozumí podíl časové složitosti nejlepšího známého sekvenčního algoritmu a takzvané paralelní práce (čas potřebný k dokončení úlohy na paralelním počítači násobený počtem procesorů). Tato definice není pro naše potřeby příliš vhodná – předpokládá, že sekvenční algoritmus se od toho paralelního liší, což v našem případě neplatí („sekvenčním“ algoritmem je vlastně paralelní spuštěný pouze na jedné výpočetní jednotce) a také vyžaduje znalost časové složitosti. My proto budeme při vyhodnocení naměřených výsledků efektivitu pro $n$ procesorů chápat jako podíl doby běhu úlohy na nejmenším počtu procesorů ku součinu doby běhu na $n$ procesorech a počtu $n$, případně ji dále vztahovat i vzhledem k další veličině, jako například počtu iterací nebo velikosti úlohy.

## 2.4 Zrychlení

Paralelní zrychlení udává prostý fakt, a to kolikrát je paralelní algoritmus rychlejší než nejlepší známý sekvenční algoritmus. Obdobně jako u efektivity se i v tomto případě zrychlení definuje pomocí složitosti, v praxi ale můžeme s výhodou použít naměřené absolutní hodnoty času, a tak tomu bude i při našich měřeních.

Testem zrychlení rozumíme situaci, kdy velikost řešené úlohy je po celou dobu testu konstantní a postupně se mění počet výpočetních jednotek, na kterých je výpočet prováděn. Spíše než absolutní hodnoty časů potřebných k dokončení výpočtu u jednotlivých běhů nás v tomto případě zajímají jejich poměry – to, jakým způsobem se výpočet chová při rostoucím počtu procesorů. V ideálním případě by zrychlení mělo růst lineárně (kolikrát zvětšíme počet procesorů, tolikrát se sníží čas výpočtu), realita má však k ideálnímu stavu obvykle daleko, a to z důvodů, které jsme si popsali v kapitolách o Amdahlovu a Gustafsonovu zákonu a které si také dále ukážeme v části věnované hardwaru paralelních systémů.

## 2.5 Škálovatelnost

Zatímco zrychlení sleduje chování stále stejné úlohy (stejných vstupních dat) při měnícím se počtu procesorů, škálovatelnost se na problém dívá z trochu jiného úhlu pohledu: S rostoucím počtem procesorů se adekvátně zvětšuje velikost úlohy. Tento přístup může být mnohem užitečnější – často je důležitější spočítat větší úlohu v přijatelném čase, než se snažit minimalizovat dobu běhu nějaké menší úlohy. Ideálně škálovatelný algoritmus by se tedy, podle výše uvedeného popisu, měl vyznačovat konstantním časem výpočtu. K testu jsou však potřeba vstupní data o požadovaných velikostech, což vždy nelze úplně přesně zajistit (a tato situace nastane i u námi prováděných testů) – problém však lze s mírným snížením přesnosti obejít použitím dat přibližné velikosti a vypočítanou hodnotu škálovatelnosti adekvátně interpolovat tak, jako by velikost dat odpovídala přesně.

# 3 Softwarové nástroje pro paralelní výpočty

V této kapitole si představíme některé softwarové prostředky, které pro nás jsou zajímavé zejména tím, že jsou (ať již přímo nebo nepřímo) použité v programu Flow123d. Většina aplikací paralelních algoritmů nachází uplatnění v oblastech jako inženýrské projektování, simulování, modelování přírodních soustav ve fyzice, biologii, chemii a podobně. Ve všech těchto oblastech vede řešení matematických modelů na soustavy lineárních rovnic, hojně se zde tedy využívá operací s vektory a maticemi. Nebudeme se zde zabývat implementačními detaily, jako je například typ datové struktury pro uložení matice v paměti, podíváme se spíše na praktickou stránku věci, na různé softwarové balíky a knihovny a jejich funkce, které nám nabízejí. Tyto knihovny se samozřejmě nemusí používat jen a pouze v paralelních systémech (je to však jejich nejčastější použití, na těch jednoprocesorových by ale fungovaly také), vedle nich si však představíme i softwarové nástroje výlučně pro paralelní prostředí. Tento náš přehled nemůžeme chápat jako absolutní představení dané problematiky, jelikož zde nepojednáváme například o čistě paralelních programovacích jazycích, jakým je třeba HPF (High Performance Fortran), nebo o komunikační knihovně PVM (Parallel Virtual Machine).

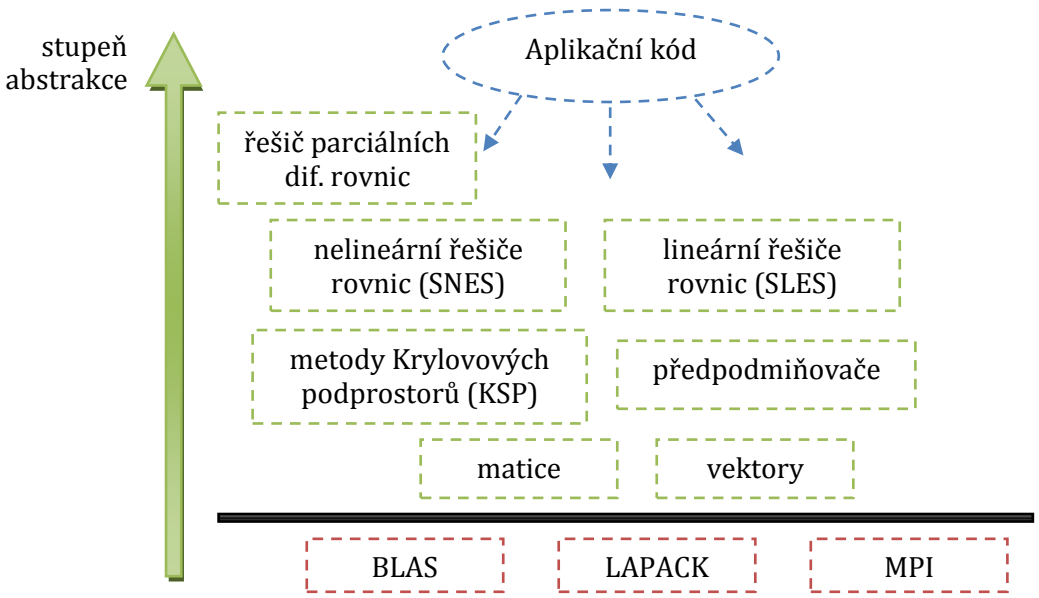
## 3.1 Knihovny BLAS a LAPACK

Knihovna BLAS (Basic Linear Algebra Subprograms) představuje nejznámější sadu funkcí pro základní operace lineární algebry. Byla napsána v roce 1979 v jazyce Fortran a od té doby se dočkala přepisu (v různých programovacích jazycích) jak pro různé počítačové architektury, tak dokonce i pro jednotlivé procesory. Původní verze je k dispozici zdarma ve formě zdrojových kódů, jednotlivé implementace však mohou být i velmi drahou záležitostí. Společným znakem všech těchto implementací je snaha o maximální možnou optimalizaci. Původní BLAS (takzvaný Level 1) obsahuje opravdu jen základní algebraické operace pro skaláry a vektory, v průběhu času se k němu přidaly i operace typu matice-vektor (Level 2) a matice-matice (Level 3).

LAPACK (Linear Algebra PACKage) jsou knihovny využívající funkce knihoven BLAS a poskytují nám například implementaci algoritmů pro řešení soustav lineárních rovnic, Choleského a Schurovu faktorizaci pro reálné i komplexní matice, pro řešení standardní i zobecněné úlohy vlastních čísel, vektorů a další maticové výpočty. Tak jako původní BLAS je základní LAPACK volně šiřitelný, stejně tak ale existují i (povětšinou draze placené) implementace optimalizované pro specifický hardware. Od společnosti Intel je k dispozici knihovna MKL (Math Kernel Library), která je vysoce optimalizovaná pro běh právě na procesorech od téže firmy. V MKL jsou implementovány vedle řady dalších funkcí i funkce z BLAS a LAPACK a tuto knihovnu je možno použít ve spolupráci s knihovnami PETSc, které si popíšeme v následující kapitole.

## 3.2 Knihovny PETSc

PETSc, neboli Portable, Extensible Toolkit for Scientific Computation je sada zdarma dostupných datových struktur a funkcí pro (paralelní) numerické řešení soustav parciálních diferenciálních rovnic. Knihovny jsou (jak jinak) založeny na funkcích z BLAS a LAPACK, k nim však přidávají onu možnost paralelních výpočtů, realizovaných zde pomocí rozhraní MPI (bude popsáno v následující kapitole). Z pohledu vývojáře je PETSc plně využitelné z programovacích jazyků Fortran, C++ a nově i Python, přičemž jeho architektura je navržena v duchu snadné rozšiřitelnosti – nalezneme zde rozhraní pro spolupráci se spoustou dalších aplikací a modulů, přičemž některé z nich je možno dokonce automaticky nainstalovat při prvotní konfiguraci. A co všechno nám knihovny PETSc vlastně nabízí? Jsou to především lineární a nelineární řešiče soustav rovnic založené na metodě Krylovových podprostorů, respektive na Newtonově metodě, dále pak několik předpodmiňovačů a různé datové struktury pro uložení řídkých matic. Jsou to ale i podrobné statistiky, které knihovny umí zapsat do výstupního souboru a které obsahují podrobný přehled o využití paměti, procesorového času či počtu volání funkcí MPI, a to nejen souhrnné za celý běh programu, ale dokonce i pro jednotlivé knihovní funkce. Jediné, co pro její použití musíme udělat, je přidání parametru `-log_summary`.




*Obr. 1: Zjednodušené schéma knihoven PETSc*

### 3.2.1 Konfigurace PETSc

Před prvním použitím je nutno PETSc knihovny nakonfigurovat. To je fáze, ve které se mimo jiné zjišťují informace o použitém systému, hledají a testují se potřebné knihovny nebo kompilátor a podobně.

Ještě před samotným spuštěním konfiguračního skriptu je zapotřebí nastavit proměnnou prostředí nazývající se PETSC_DIR, která představuje cestu k adresáři s uloženými zdrojovými (a později i zkompilovanými) soubory. Díky tomu můžeme mít na jednom počítači více verzí knihoven PETSc a pomocí této proměnné mezi nimi přepínat. Další proměnnou, kterou však manuálně nastavovat nemusíme (v tom případě se použije implicitní hodnota), je PETSC_ARCH. Tato proměnná specifikuje jméno konfigurace a s její pomocí můžeme mít více různých konfigurací od stejné verze PETSc. V praxi a při testování (což byl i náš případ) máme obvykle nainstalovanou pouze jednu verzi PETSc (PETSC_DIR odkazuje stále na stejný adresář) a několik připravených konfigurací, mezi kterými se snadno přepneme přepsáním oné proměnné PETSC_ARCH a následným přeložením knihoven. Ve vlastním kódu naší aplikace tak není potřeba cokoliv měnit.

Vlastní konfigurační skript může přebírat řadu parametrů (žádný z parametrů není povinný), jimiž můžeme specifikovat použitý kompilátor, umístění knihoven BLAS a LAPACK (pokud máme k dispozici verzi optimalizovanou pro použitý hardware), zda používat sdílené knihovny (snižuje se tím velikost výsledných binárních souborů), vypnout či zapnout ladicí informace a dále také určit jaké externí knihovny budeme chtít použít (ty se automaticky stáhnou z internetu a nakonfigurují).

## 3.3 MPI

MPI (Message Passing Interface) představuje specifikaci standardu pro podporu paralelního řešení výpočetních problémů u počítačových clusterů. Konkrétně se jedná o API založené na zasílání zpráv mezi jednotlivými uzly, a to jak v režimu point-to-point (komunikace mezi dvěma uzly), tak i jako tzv. broadcast (globální vysílání všem uzlům). Ve specifikaci jsou zahrnuty jak architektury se sdílenou pamětí, tak s pamětí distribuovanou (dnes častější). Z pohledu referenčního modelu ISO/OSI je MPI posazeno do páté, tedy relační vrstvy, přičemž většina implementací používá jako transportní protokol TCP.

Při návrhu celého rozhraní i při jeho implementaci byl vždy kladen důraz především na výkon, škálovatelnost a přenositelnost. Klíčovou vlastností, která může být jak nevýhodou, tak i výhodou, je její nízkoúrovňový přístup. Nehodí se tedy pro rychlý vývoj aplikací (RAD), ale spíše pro aplikace, kde je rozhodující rychlost běhu, což je ale pro paralelní systémy typické. To je pravděpodobně i důvodem, proč se stala v této oblasti de-facto standardem.

První návrh standardu MPI pochází z roku 1994, v současné době existují dvě hojně používané verze: starší verze 1.2 (často označovaná pouze jako MPI-1), která klade důraz na posílání zpráv, a verze 2.1 (označovaná MPI-2), která mimo jiné přináší navíc paralelní vstupně výstupní operace, dynamickou správu procesů a vzdálené operace s pamětí. Jazyk, ve kterém je implementace MPI realizována, nemusí být tím samým, v jakém je napsána vlastní aplikace. Nejčastěji se však setkáme s implementací v C, C++ či jazyce symbolických adres a výjimkou není ani podpora přímo na úrovni hardwaru. Je vidět, že implementací existuje celá řada, ta

první a původní se jmenuje MPICH a existuje jak pro starší, tak i novější verzi MPI. Velmi používanou implementací je OpenMPI – jde o poměrně nový, avšak slibný projekt, který se pokouší o spojení několika ostatních implementací (FT-MPI, LA-MPI, LAM/MPI a PACX-MPI) s tím, že z každé implementace si vezme to, v čem je nejlepší. Vznikla tak kompletně nová implementace MPI-2. Za jejím vývojem stojí celé konsorcium firem a univerzit, mezi něž patří například University Of British Columbia, Cisco, IBM nebo Sun Microsystems.

## 3.4 Dávkový systém

Při spouštění úloh na paralelním počítači je velmi vhodné mít nějaký systém, který úlohy spravuje, to jest řadí příchozí požadavky do fronty a na základě předem definovaných pravidel je z fronty vybírá a spouští. Takový plánovač úloh (angl. job scheduler) přitom musí mít přehled o vytížení jednotlivých výpočetních uzlů a jejich procesorů a zároveň zajistit, aby na jednom procesoru neběželo více úloh současně, což by zbytečně snižovalo výkon. Pokud bychom nevyužili plánovače úloh a naše aplikace spouštěli klasickým způsobem (tzv. interaktivně), velmi pravděpodobně by brzy nastala právě ta situace, kdy by některé z jader procesorů obstarávalo více než jednu úlohu, zatímco by zde stále byla další nevyužitá jádra. Tímto způsobem lze tedy poměrně snadno deklasovat výkon prakticky libovolného systému.

Správně nakonfigurovaný plánovač úloh má informace o použitém hardwarovém vybavení, především pak o tom, jakými procesory jsou osazeny výpočetní uzly i o počtu jejich jader. Díky tomu může spouštěným úlohám strategicky přidělovat jádra a procesory tak, aby se minimalizovala nutnost přenosu dat přes pomalé komunikační kanály (např. propojovací ethernetovou síť). Tato strategie je obvykle velmi žádaná, není však univerzální – u některých, typicky paměťově velmi náročných aplikací, může dojít až k pádu aplikace vlivem nedostatku paměti.

Na trhu existuje několik dávkových systémů, a to jak komerčních, tak i zdarma dostupných. V tomto odstavci si stručně popíšeme dva z nich – PBS a SGE, a to hlavně z toho důvodu, že to jsou dávkové systémy použité na dvou testovacích počítačích Hydra a Rex, o kterých bude řeč v následující kapitole. Oba systémy jsou

si do jisté míry podobné[5] co se týče používaných příkazů a uživatel tak při běžné práci ani nepotřebuje vědět, který ze systémů právě používá. PBS (Portable Batch Systém) je starším systémem, jehož stále vyvíjená a podporovaná verze je komerční (PBS Professional), v minulosti z ní však byla oddělena varianta s otevřeným zdrojovým kódem OpenPBS, která se v současnosti sice již neudržuje, vychází z ní však stále čile vyvíjená verze Torque. Oproti tomu mladší SGE (Sun Grid Engine) byl od počátku vyvíjen jako open-source projekt (i když existovala i jeho placená verze). Po akvizici firmy Sun společností Oracle v roce 2010 byl SGE přejmenován na OGE, ve kterém se opět zrcadlí název společnosti. Navzdory tomuto přejmenování se v této práci budeme držet staršího SGE, jakožto zavedeného názvu (ve všech publikacích a zdrojích, i nově vydaných, se stále používá starší název) a také proto, že na clusteru Hydra je nainstalována právě ona starší verze.

Pro spuštění úlohy pomocí dávkového systému je zapotřebí vytvořit textový soubor, ve kterém teprve specifikujeme, co vlastně chceme spustit, a případně uvedeme i některé další parametry. Formát těchto souborů není obecně nijak standardizován a liší se systém od systému. Jejich jméno může být libovolné, obvykle se ale používá název s příponou, která je odvozena od názvu příkazu, kterým se úlohy předávají ke zpracování dávkovému systému, tedy `qsub`. Proto se na tyto soubory budeme nadále v našem textu odkazovat jako na `qsub` soubory.

# 4 Paralelní architektury

Paralelní počítače prošly v posledních dvou třech dekádách bouřlivým rozvojem (a stále jím procházejí), ve kterém však lze vypozorovat jeden důležitý trend, a tím je odklon od drahých specializovaných a většinou na zakázku vyráběných hardwarových součástek či celých systémů. Řada teoreticky přitažlivých řešení se v praxi nedokázala prosadit, a tak na trhu z tohoto důvodu začínají převládat cenově dostupné clustery[1] – paralelní svazky počítačů propojených mezi sebou například klasickou ethernetovou sítí a vybavené standardními operačními systémy, jejichž jednotlivé uzlové počítače mohou být jak výkonné pracovní stanice, tak rovněž i levnější osobní počítače.

## 4.1 Paralelismus

Pokud se chceme zabývat paralelními počítači, musíme nutně začít u pojmu paralelismus jako takového. Ten by se dal přeložit slovem souběžnost a v našem kontextu představuje schopnost počítače či systému obsluhovat více aplikací „současně“.

### *4.1.1 Pseudoparalelismus*

Procesor, respektive jádro procesoru počítače dokáže v jednu chvíli vykonávat pouze jeden proces, což se může zdát jako rozpor mezi tím, co známe z běžného života, kde nám v operačním systému běží „současně“ několik aplikací. Ve skutečnosti zde žádný rozpor není – operační systém řeší tuto situaci (málo procesorů, spousta procesů) tak, že každému procesu přidělí CPU jen na krátkou chvíli, což v uživateli vyvolá dojem, že aplikace běží současně. Nicméně CPU vykonává vždy pouze jeden proces, a proto se této technice říká pseudoparalelismus (nepravý paralelismus).

[1] V porovnání 500 nejvýkonnějších počítačů světa podle http://www.top500.org z listopadu 2010 mají clustery podíl téměř 83% a přes 44% z nich používá jako propojovací médium gigabitový ethernet.

### 4.1.2 Paralelní systém

Paralelní systém již umožňuje zpracování několika úloh opravdu paralelně. Zatímco v minulosti jsme úlohou mohli rozumět i jednotlivou instrukci (v tom případě by ale všechny dnešní procesory byly paralelní, jelikož umožňují tzv. pipelining), dnes se tím rozumí spíše celé programy, aplikace, a tak budeme paralelní systém chápat i my ve zbylé části textu. Nutnou podmínkou pro paralelní systém je tedy přítomnost více výpočetních jednotek, na kterých nebude docházet k přepínání procesů popsaných výše v odstavci o pseudoparalelismu.

Je také dobré si uvědomit, jak tyto paralelní systémy fungují a jakým způsobem vlastně aplikace dosahují paralelismu. V klasickém stylu programování se paralelismu dosahuje prostřednictvím vláken (takzvaný middle-grained paralelismus), ta jsou však použitelná jen u systémů se sdílenou pamětí, jelikož sdílejí stejný adresní prostor. Oproti tomu zde se používá metoda, kdy na všech procesorech je spuštěn naprosto stejný kód a teprve jejich vzájemnou komunikací (prostřednictvím MPI rozhraní kupříkladu) konkrétní procesor zjistí, jakou činnost má nad vstupními daty vykonávat. Tato metoda je často označována jako coarse-grained paralelismus.

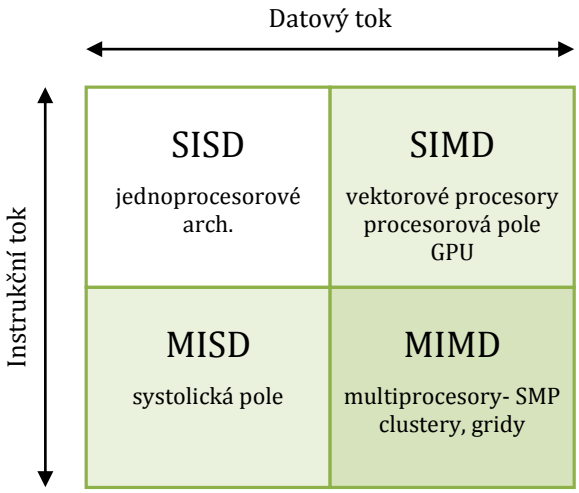
## 4.2 Taxonomie

Širší paleta architektur si vynutila podrobnější dělení, taxonomii, na kterou se podíváme v této části, a to včetně popisu nejdůležitějších vlastností. Nebudeme se zde vůbec zabývat paralelními počítači založenými na funkcionálních modelech, logických modelech či neuronových sítích, zůstaneme, a to i ve zbylé části textu, výhradně v oblasti imperativních počítačů von Neumannova typu.

Nejběžnějšími způsoby dělení počítačových architektur, kterými se zde budeme krátce věnovat, je dělení z pohledu toků instrukcí a dat (takzvaná Flynnova taxonomie) a dále pak dělení podle organizace paměti. Lze se však samozřejmě setkat i s jinými kritérii pro „škatulkování“, jakým může být například způsob propojení jednotlivých výpočetních uzlů.

### 4.2.1 Flynnova taxonomie

V roce 1966 vytvořil Michael J. Flynn dělení počítačových architektur do čtyř skupin (SISD, SIMD, MISD, MIMD)[2] na základě toho, kolik v dané architektuře existuje datových a instrukčních proudů.




*Obr. 2: Flynnova taxonomie paralelních systémů*

- SISD – dnes již málo častý typ obvykle jednoprocesorových systémů, které sice mohou za určitých situací zpracovat více instrukcí či dat současně (to je případ takzvaného pipeliningu), to je ale považováno spíše za paralelní zpracování sekvenčního kódu než za provedení paralelního kódu
- SIMD – jednou instrukcí umožňují zpracovat více dat (např. sečíst dva vektory), běžné u vektorových procesorů či procesorů s instrukcemi MMX, SSE rozšiřujících architekturu x86
- MISD – velmi zřídka používaný typ se kterým se lze setkat u speciálních architektur, jako jsou systolická pole a podobně, kde jsou jedna data postupně zpracovávána více instrukcemi
- MIMD – obsahují více výpočetních jednotek, tudíž se jedná o víceprocesorové systémy či vícejádrové procesory, které umožňují nezávisle na sobě asynchronně provádět různé instrukce nad různými daty

[2] Anglický název lze dešifrovat podle klíče: S = single, M = multiple, I = instruction, D = data

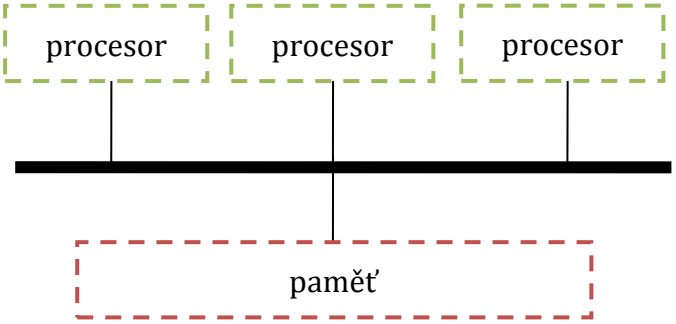
Z výše uvedeného výčtu je možná patrná hlavní a poměrně velká nevýhoda Flynnovy taxonomie – dnes, po zhruba 35 letech od jejího vzniku, jsou téměř všechny paralelní počítače jednoho typu, MIMD. Na druhou stranu však představuje stále hojně používaný způsob dělení počítačových architektur, a proto jsme ji v tomto textu zmínili i my.

### *4.2.2 Taxonomie z hlediska organizace paměti*

Jiný pohled (oproti Flynnově taxonomii) na počítačové architektury nám přináší rozdělení na ty se sdílenou pamětí a distribuovanou pamětí, ke kterým se zvlášť v poslední době často přidává i jejich kombinace, architektury se sdílenědistribuovanou pamětí.

#### 4.2.2.1 Sdílená paměť

Systémy se sdílenou pamětí představují situaci, kdy jsou procesory (výpočetní jednotky) připojeny k jedné globální paměti. Ať už je sdílená paměť realizována hardwarově nebo softwarově (v tomto případě může být fyzicky distribuovaná, ale programům se jeví pouze jako jedna, globální), z pohledu programátora je velmi pohodlné a poměrně jednoduché ji využívat – o její správu a konzistenci se totiž stará operační systém. Na druhou stranu mohou být systémy se sdílenou pamětí hůře škálovatelné, méně flexibilní a při větším počtu procesorů může docházet k degradaci výkonu způsobené vysokým počtem přístupů do relativně pomalé paměti. Nejčastější variantou systému se sdílenou pamětí jsou takzvané UMA (Uniform Memory Access) – unifikované proto, že přístupová doba libovolného procesoru k hlavní paměti je konstantní, nezávisle na tom, který z procesorů o data žádá a ve kterém paměťovém modulu jsou data uložena. Vedle sdílené hlavní paměti může mít každý procesor svou vlastní rychlou vyrovnávací paměť cache. Nejběžnějším způsobem hardwarové realizace je vzájemné propojení procesorů přes jednu sběrnici, ke které je připojena i globální paměť. Toto řešení (znázorněno na obrázku 3) se nazývá SMP (Symmetric Multiprocessing) a běžně se s ním lze setkat v klasických osobních počítačích.

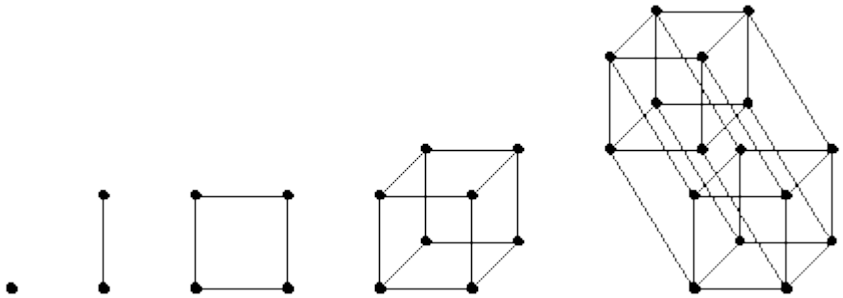



*Obr. 3: Sdílená paměť - SMP*

#### 4.2.2.2 Distribuovaná paměť

U systémů s distribuovanou pamětí má každý procesor vlastní individuální paměťový prostor a jednotlivé procesory „neví“ nic o pamětech ostatních procesorů – veškerá výměna dat mezi nimi tak musí probíhat prostřednictvím mechanismu zasílání zpráv. Ačkoliv se pojmem distribuovaná paměť primárně myslí logické rozdělení paměti, v drtivé většině případů je paměť distribuovaná i fyzicky. Jelikož se k datům v paměti přistupuje vždy „z jednoho místa“, nevznikají zde kolize a správa paměti je tak mnohem jednodušší. Aby bylo posílání zpráv mezi procesory co nejrychlejší, bylo by v ideálním případě potřeba vzájemně propojit výpočetní jednotky ve stylu každý s každým – to je v praxi (při větších počtech procesorů) samozřejmě krajně nepraktické, a proto má každý procesor přímé spojení pouze s několika dalšími. Jedním z oblíbených způsobů, jak toto propojení provést, je například propojení ve tvaru takzvané hyperkrychle (obrázek 4). Jelikož je možné předávat zprávy pouze mezi sousedními procesory, je zřejmé, že komunikace mezi těmi vzdálenými může být pomalá.

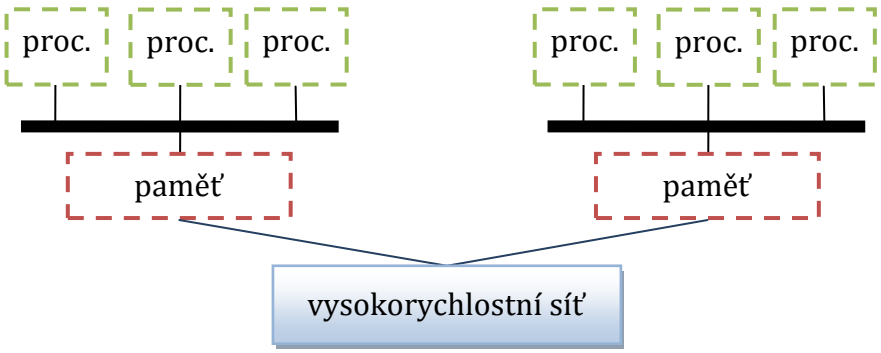
Významným důsledkem použití architektury s distribuovanou pamětí je omezení paměti, kterou má proces k dispozici – dostupná je mu pouze ta, kterou disponuje uzel, na němž konkrétní proces běží. U paměťově náročných aplikací tak mohou nastat chyby nedostatku paměti i v případě, že celkové paměti je dostatek, avšak požadavek na některém uzlu překročil daný limit. Tyto chyby se u architektur se sdílenou a sdílenědistribuovanou pamětí projeví až při vyčerpání veškeré paměti.



*Obr. 4: Ukázka propojení typu hyperkrychle pro jeden až 16 procesorů*

#### 4.2.2.3 Sdílenědistribuovaná paměť

Tento typ architektury je znám pod zkratkou NUMA (Non-Uniform Memory Access) a představuje kombinaci obou výše zmíněných typů. Základní myšlenka vychází z předpokladu, že jednak je nutné minimalizovat absolutní počet přístupů do paměti, aby nedocházelo k čekání, pokud několik procesorů žádá data zároveň, za druhé je pak zapotřebí snížit komunikaci se „vzdálenými" procesory či paměťmi. NUMA tento požadavek řeší způsobem zobrazeným na obrázku 5: Oproti SMP, kde jsou všechny procesory připojeny k jediné sběrnici, je zde tento počet procesorů omezen – existuje zde tedy více paměťových sběrnic (mezi sebou propojených vysokorychlostní sítí). Procesory tudíž dokážou velmi rychle přistupovat k datům v blízké paměti, základní úkolem je tedy zajistit, aby se požadovaná data v této paměti nacházela a nemusela se získávat z pomalejší vzdálené paměti.




*Obr. 5: Schéma NUMA architektury*

## 4.3 Paralelní systémy v praxi

Pro paralelní systémy se kvůli jejich vysokému výkonu obecně vžilo označení „superpočítače“. Samozřejmě neexistuje přesná hranice, kdy je možné počítač označit za superpočítač, obvykle se však hovoří o minimálně desetinásobně vyšším výkonu oproti běžně dostupným počítačům. V minulosti superpočítače představovaly pouze na zakázku vyráběné stroje, které v té době byly technologickou špičkou, čemuž však odpovídala i cena. I proto dnes začínají převládat svazky „klasických“ počítačů – clustery.

Cluster je v podstatě pouze seskupení propojených počítačů, které spolu úzce spolupracují, takže se navenek mohou tvářit jako jeden počítač. Jednotlivé počítače tvořící cluster (takzvané uzly) mohou být navzájem po hardwarové stránce i poměrně odlišné a mohou je tvořit například i klasické stolní počítače, obvykle je však sestaven podobně jako většina serverů – umístěním počítačů do speciální skříně, takzvaného racku. Hlavní výhoda clusterů spočívá především v jejich ceně – jsou levnější, než by byl jediný počítač o srovnatelné rychlosti, spolehlivosti nebo jiných sledovaných parametrech. Požadovaná funkce clusteru se odráží i ve způsobu, jakým je můžeme klasifikovat – běžné jsou výpočetní clustery, clustery s vysokou dostupností, s rozložením zátěže nebo například úložné clustery.

Oproti clusterům, u kterých jsou počítače propojeny obvykle poměrně rychlou sběrnicí či sítí (ať již nějakou proprietární technologií nebo například klasickým gigabitovým ethernetem), jsou počítače u takzvaných gridů spojeny mnohem volněji. Obvykle se jedná o spojení na mnohem větší vzdálenosti a nesrovnatelně pomalejšími technologiemi – klasickou ukázkou může být například projekt SETI@home, který propojuje počítače na celém světě prostřednictvím sítě Internet. Může se do něj zapojit kdokoliv a výpočetní výkon připojených počítačů je posléze využit k hledání mimozemské inteligence.

Samozřejmě bychom mohli najít i další typy a architektury superpočítačů, popsali jsme si pouze ty z našeho pohledu nejzajímavější a zároveň nejvíce rozšířené. I v dnešní době však lze nalézt superpočítače postavené na míru

a optimalizované pouze pro specifický okruh úloh – jedním z nejznámějších je počítač pojmenovaný Deep Blue, vytvořený pro hraní šachu.

## 4.4 Superpočítače Hydra a Rex

Měření, která jsou uvedena v této práci, jsme (až na některé výjimky) vždy prováděli na dvou počítačích, pojmenovaných Hydra a Rex[3].

Hydra je výpočetní cluster Fakulty mechatroniky, informatiky a mezioborových studií Technické univerzity v Liberci a fyzicky se skládá ze dvou typů uzlů: Je to jednak 12 uzlů od firmy Dell osazených vždy dvěma dvoujádrovými procesory Intel Xeon a dále pak 17 uzlů společnosti Sun ve kterých lze nalézt dva jednojádrové procesory AMD Opteron. Každý uzel disponuje 4GB paměti RAM a mezi sebou jsou propojeny sítí ethernet, operačním systémem je Linux CentOS. Pokud se budeme potřebovat nadále v této práci odkazovat na jeden či druhý typ těchto výpočetních uzlů, budeme pro zjednodušení hovořit pouze o uzlech Dell či uzlech Sun.

Druhý počítač, nazvaný Rex, je superpočítač Centra pro intenzivní výpočty na ČVUT v Praze a celkem jej tvoří 96 procesorů Intel Itanium. Nejdůležitějším rozdílem vůči clusteru Hydra je jeho odlišná architektura – oproti distribuované paměti u Hydry využívá Rex paměť sdílenědistribuovanou (NUMA, celkem je jí k dispozici na 270 GB). To nám může v jejich vzájemném srovnání (popsaném v kapitolách 5 a 7) přinést zajímavé výsledky, jelikož Rex sice disponuje procesory o nižší hodinové frekvenci, díky použité architektuře ale můžeme očekávat lepší škálovatelnost úloh.

[3] Podrobnější technické specifikace jsou uvedeny v příloze

# 5 Test ukázkových úloh z knihoven PETSc

Abychom ověřili platnost Amdahlova, respektive Gustafsonova zákona a dostali alespoň rámcovou představu o tom, jaké výsledky můžeme v budoucnu očekávat, rozhodli jsme se otestovat škálovatelnost a zrychlení na některé z úloh, které jsou pro demonstrační účely součástí knihoven PETSc[4]. Tento test je také vhodný pro prvotní porovnání obou superpočítačů, na kterých měření provádíme.

Knihovny PETSc byly ve všech případech konfigurovány bez ladicích informací (`--with-debug=no`). Hodnoty vynesené v grafech jsou vždy aritmetickým průměrem tří měření, abychom vyloučili vzniklé odchylky. Ani jeden z testovaných počítačů totiž není kompletně osazen naprosto stejnými procesory, a pokud přesně nespecifikujeme požadované uzly, je výpočet vždy ovlivněn tím, jak dávkový systém výpočet rozdělí. Použité úlohy vykazují na různém počtu procesorů různý počet iterací, a jelikož nás zajímá spíše škálovatelnost matematických operací (zejména maticového násobení) nežli škálovatelnost celé úlohy, budeme naměřené hodnoty odpovídajícím způsobem přepočítávat na efektivitu iterací, a tím v podstatě testovat efektivitu maticového násobení – samozřejmě pouze v případech, kdy to má smysl, neboli při testech zrychlení a škálovatelnosti, ale již ne v případech, kdy budeme sledovat a porovnávat absolutní čas. Pro přehlednost zde neuvádíme všechny naměřené hodnoty, ale pouze grafy z nich sestrojené, kompletní tabulky, ze kterých jsme vycházeli, jsou uvedeny v příloze.

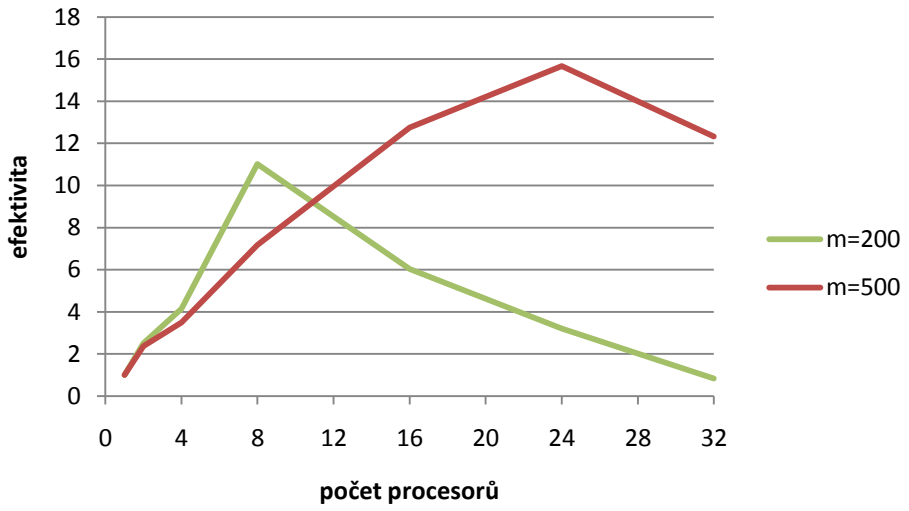
## 5.1 Úloha ex3

Jako první jsme zvolili úlohu s názvem ex3. Tato úloha paralelně řeší lineární soustavu rovnic prostřednictvím lineárního řešiče (KSP), založeného na iterační metodě Krylovových podprostorů. Jediným vstupním parametrem úlohy je *m*, určující velikost řešené čtvercové sítě. Aby se otestovalo paralelní skládání řešené matice, je vytvořená matice záměrně rozložena mezi procesory odlišným způsobem, než kterým byla sestavena.

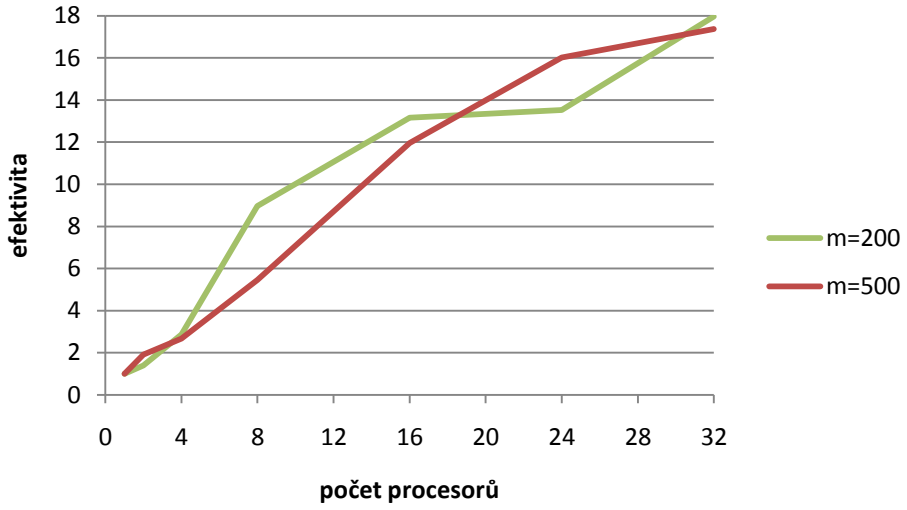
[4] Námi použité úlohy jsou uloženy v adresáři s nainstalovanými knihovnami PETSc ve složce src/ksp/ksp/examples/tutorials

Hned úvodní měření však ukázala, že naměřené časy jsou v rozporu s tím, co bychom očekávali a zrychlení při použití více procesorů roste mnohem rychleji, dochází tu k takzvanému superlineárnímu zrychlení. Toto chování je s největší pravděpodobností způsobeno vyrovnávacími pamětmi cache v procesoru spolu se situací, kdy se celá řešená úloha nebo alespoň její velká část vejde do těchto velmi rychlých pamětí, čímž se výpočet velmi urychlí. Pokud se této situace v praxi podaří dosáhnout, je samozřejmě velmi vhodné jí využít. V našem případě jsme však očekávali, že zbytek testovacích úloh (jak úloh z knihoven PETSc, tak později i pro program Flow123d) se nebude chovat takto ideálně, a proto jsme se rozhodli zvolit pro test ještě jednu úlohu obsaženou v instalaci PETSc. Touto úlohou je ex12, o které bude pojednávat následující kapitola.

Níže uvedený graf 1 zachycuje situaci na clusteru Hydra. Pro obě velikosti úlohy je patrné, že zde existuje jisté maximum – bod, kde dochází ke zlomu a efektivita dále již klesá, což je rovněž i místo, kde začne převládat komunikační režie nad výhodami plynoucími z rychlých pamětí cache. U měření na počítači Rex (graf 2) jsme sice takové chování nepozorovali, lze však předpokládat, že pro větší počet procesorů podobný zlomový bod, respektive maximum, bude existovat také.




*Graf 1: Test zrychlení pro dvě velikosti úlohy ex3 na clusteru Hydra*




*Graf 2: Test zrychlení pro dvě velikosti úlohy ex3 na počítači Rex*

## 5.2 Úloha ex12

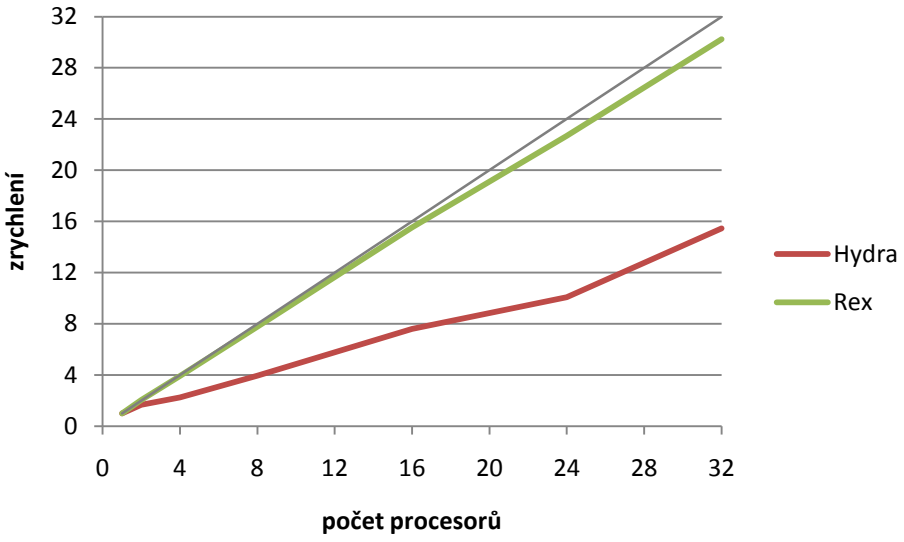
Úloha ex12 řeší podobně jako ex3 soustavu lineárních rovnic za pomocí KSP, vstupem jsou dva parametry *m* a *n* určující velikost řešené sítě. V případě ex12 se však na rozdíl od ex3 používá předpodmiňovač, kterým je Jacobiho metoda. Předpodmiňovačem obecně rozumíme nějakou transformaci, s jejíž pomocí převedeme daný problém do podoby, která je vhodnější pro numerické řešení. Jak se však ukázalo, použitý předpodmiňovač způsobuje, že výpočet nekonverguje (je nastaveno automatické ukončení výpočtu po 10000 iteracích). Takové chování úlohy je pro porovnání výkonu samozřejmě absolutně nevhodné a je potřeba ho změnit. Knihovny PETSc nabízí možnost pomocí parametru `pc_type` předat úloze námi zvolený předpodmiňovač a pomocí parametru `sub_pc_type` specifikovat předpodmiňovač pro subdomény. Námi požitý příkaz při spouštění úlohy ex12 tedy ve výsledku vypadal takto:

```
ex12 -m 2000 -n 2000 -pc_type asm -sub_pc_type ilu
```

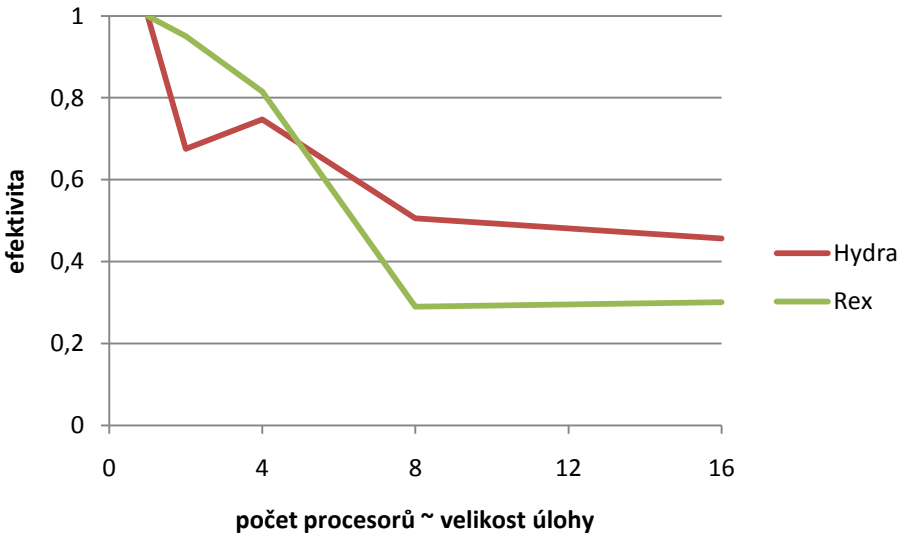
Pokud použijeme výše uvedený předpodmiňovač asm (založený na aditivní Schwarzově metodě), výpočet úlohy začne konvergovat a úloha je tak pro naše účely vhodná. Dále již tedy budou všechna naše měření prováděna pouze na této úloze.

## 5.3 Test zrychlení a škálovatelnosti

Nejdůležitějšími testy pro porovnání superpočítačů jsou testy zrychlení a škálovatelnosti. Testem zrychlení (graf 3) rozumíme situaci, kdy velikost řešené úlohy je po celou dobu testu konstantní a postupně se mění počet procesorů. Oproti tomu při testu škálovatelnosti (graf 4) roste velikost úlohy přímo úměrně s počtem procesorů, na kterých se výpočet provádí. Konkrétně v našem případě používáme vždy čtvercové sítě, a to pro běh na jednom procesoru o velikosti strany 500 bodů. Jelikož velikost sítě roste kvadraticky, pro běh na 16 procesorech tudíž vychází čtverec o velikosti strany 2000 bodů.




*Graf 3: Test zrychlení pro úlohu ex12 [m=2000; n=2000]*




*Graf 4: Test škálovatelnosti pro úlohu ex12*

Z uvedených grafů jsou patrné rozdíly mezi oběma testovanými počítači, které jsou z největší části dané jejich odlišnou architekturou; zatímco Rex používá sdílenědistribuovanou paměť, komunikace u Hydry probíhá přes ethernetovou síť. Poněkud zarážející je fakt, že ačkoliv má Rex při testu zrychlení výbornou efektivitu a úloha tak vykazovala téměř lineární zrychlení, při testu škálovatelnosti a postupném růstu velikosti problému je z nám neznámého důvodu jeho efektivita horší než u clusteru Hydra.

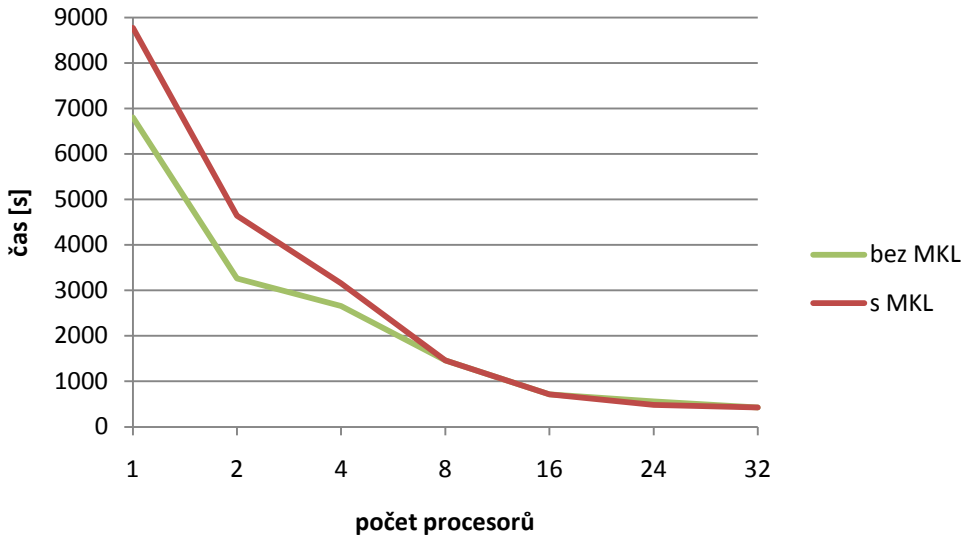
V těchto testech jsme sledovali pouze efektivitu a zrychlení vztažené k počtu iterací. Co z grafů patrné není, jsou absolutní hodnoty časů pro jednotlivá měření, ze kterých je zřejmé, že cluster Hydra je díky rychlejším procesorům o něco výkonnější než Rex. Jak jsme si však ukázali výše, výkon procesorů nemá na efektivitu žádný vliv.

## 5.4 Použití knihoven MKL a překladače firmy Intel

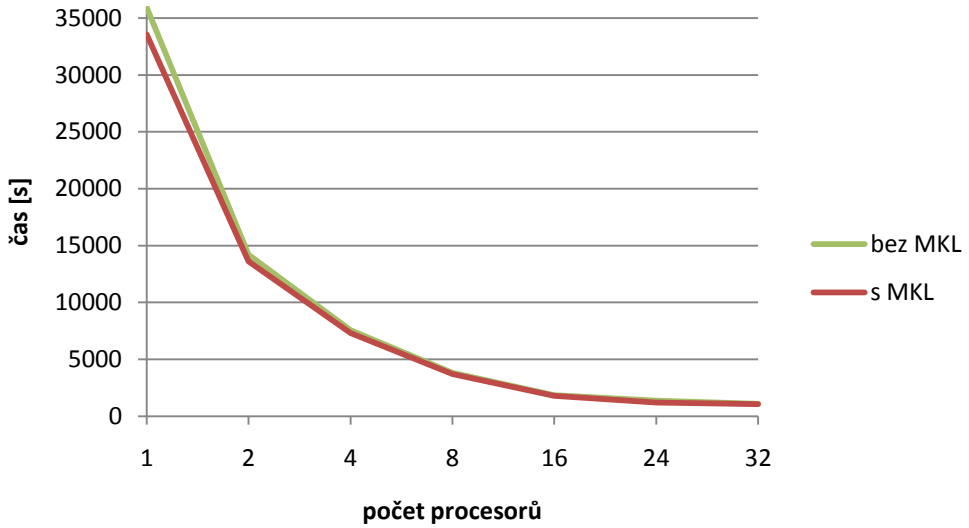
Použití BLAS a LAPACK funkcí z knihovny Intel MKL namísto těch standardních (automaticky instalovaných při konfiguraci PETSc) může poskytnout zajímavé srovnání a právě na něj se v této části zaměříme. Pro kompilaci PETSc je rovněž možné (ale ne nezbytné) použít překladač icc[5], který by měl přinést na procesorech Intel další zrychlení. Jelikož na clusteru Hydra nebyl v době psaní práce překladač icc funkční, provedli jsme měření pouze s připojenými knihovnami MKL (při překladu jsme použili parametr `--lib_blas_dir`).

Při těchto měřeních nás nezajímají ani tak hodnoty zrychlení, jako spíše čas potřebný k výpočtu, který porovnáme s odpovídající dobou běhu originální úlohy. Hodnoty na osách *x* v grafech nejsou uvedeny ve správném měřítku, je to však v zájmu zachování přehlednosti grafu.

[5] Verze překladače icc na počítači Rex: 11.1; Původně použitý překladač gcc se na obou testovaných počítačích nacházel ve stejné verzi, a to 4.1.2

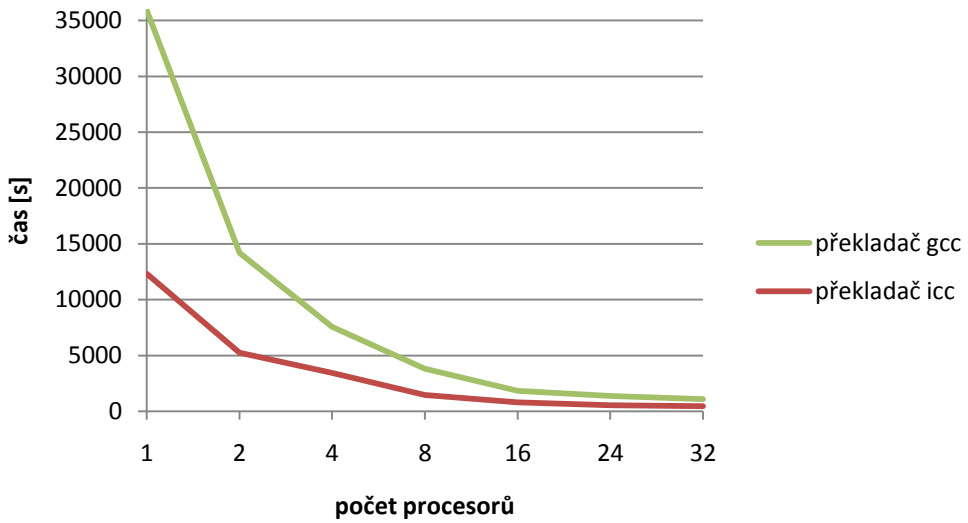



*Graf 5: Porovnání PETSc knihoven s a bez MKL – Hydra – ex12 [m=2000; n=2000]*




*Graf 6: Porovnání PETSc knihoven s a bez MKL – Rex – ex12 [m=2000; n=2000]*

Grafy ukazují, že používat pouze připojené knihovny MKL nemá příliš význam – rozhodně ne na clusteru Hydra, u kterého došlo k citelnému zpomalení (patrně vlivem procesorů AMD na některých uzlech), a diskutabilní je i přínos na počítači Rex, kde se zrychlení projevilo pouze u úloh běžících na jednom nebo dvou procesorech.




*Graf 7: Použití překladače icc od firmy Intel – ex12 [m=2000; n=2000]*

Oproti pouhému připojení knihoven MKL má použití překladače od firmy Intel daleko větší vliv a na počítači Rex jsme dosáhli zhruba třikrát lepších časů. To je především díky tomu, že jsou zde použity procesory Intel Itanium2, jejichž výkon je silně závislý na správné optimalizaci na straně překladače. Bohužel, jak již bylo uvedeno, překladač nebyl funkční na clusteru Hydra – bylo by zajímavé porovnat, zda má jeho použití vliv i na procesory od firmy AMD.

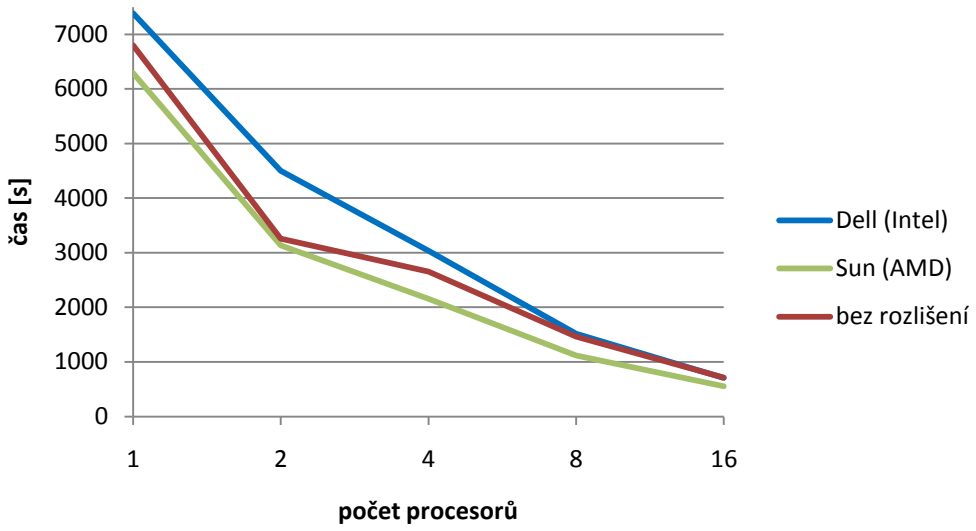
## 5.5 Porovnání Sun a Dell uzlů na clusteru Hydra

Na clusteru Hydra jsou výpočetní uzly složeny buď z procesorů AMD (uzly od firmy Sun) nebo z procesorů firmy Intel (uzly Dell). Je tedy možné porovnání rychlejších Procesorů AMD Opteron oproti sice pomalejším, ale dvoujádrovým procesorům Intel Xeon.

K tomu však potřebujeme nějakým způsobem sdělit plánovači úloh, na kterých konkrétních uzlech chceme danou úlohu spustit. To lze zařídit například parametrem `-q` příkazu `qsub`, za kterým uvedeme názvy front, do kterých chceme úlohu odeslat. Názvy těchto front jsou ve tvaru `all.q@compute-X-#.local`, ve kterém *X* je buď 0 (pro uzly Dell) nebo 1 (pro uzly Sun) a # značí pořadové číslo uzlu. Lze rovněž používat i zástupné znaky, abychom nemuseli všechny fronty vypisovat ručně, příkaz pro odeslání úlohy na všechny uzly Dell tedy může vypadat následovně:

```
qsub -pe orte 4 -q all.q@compute-0-\*.local uloha.qsub
```

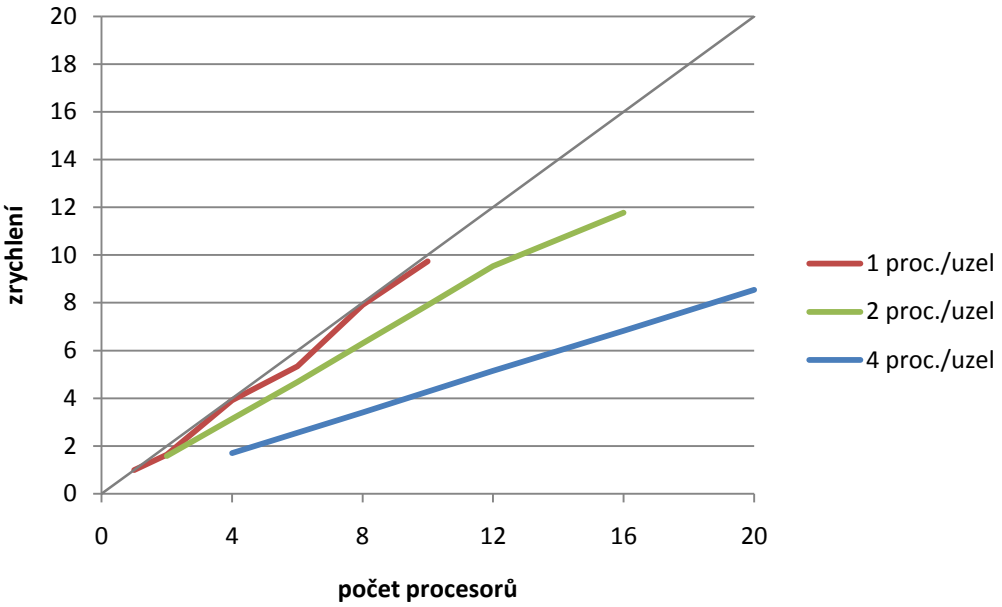
Společně s výstupem úlohy je možné vypsat i informace o tom, jaké uzly byly pro úlohu alokovány a přesněji i kolik procesorů na každém uzlu. K tomu nám dobře poslouží parametry `--display-map` a `--display-allocation`, které nám poskytuje MPI a které uvedeme ve spouštěcím `qsub` souboru.



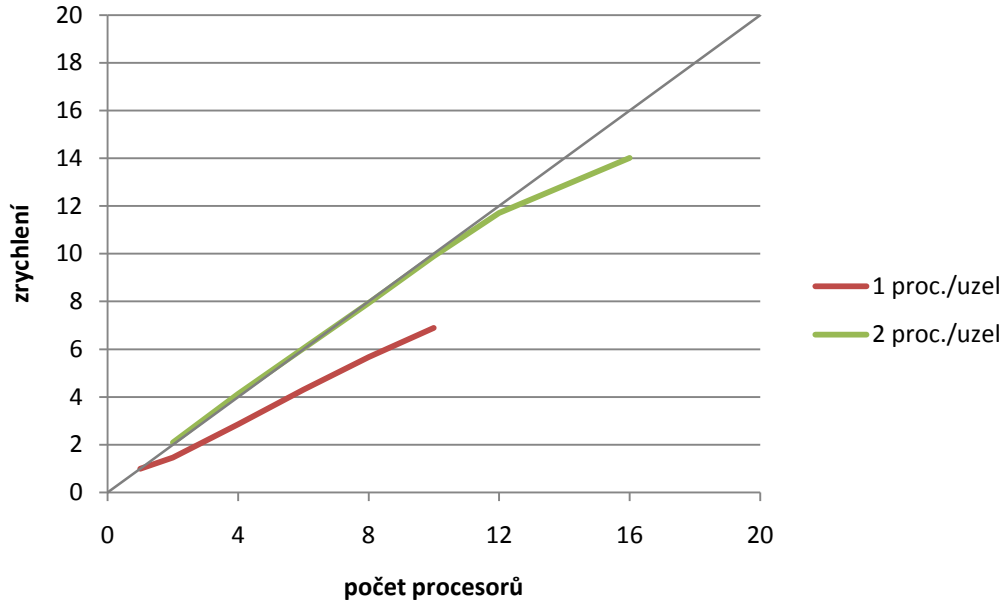

*Graf 8: Rozdíly mezi uzly Dell a Sun na clusteru Hydra – ex12 [m=2000; n=2000]*

Z grafu 8 je patrný vyšší výkon procesorů AMD na uzlech Sun, i když s rostoucím počtem procesorů se rozdíly stávají poměrně zanedbatelné. Zcela podle očekávání se pak chová měření, ve kterém nespecifikujeme, na kterých uzlech úloha poběží – hodnoty leží mezi příslušejícími hodnotami pro měření na jednotlivých uzlech.

Podle P. Šidlofa, J Horáčka a J. Staňka [4] závisí časy a efektivita nejen na tom, na jakých uzlech bude úloha spuštěna, ale i na tom, kolik procesorů, potažmo jejich jader, bude na každém uzlu úloze vyhrazeno. Abychom toto otestovali i na naší úloze, je třeba dávkovému systému zadat, kolik procesorových jader má na každém uzlu pro výpočet vyhradit. Systému SGE toto nelze zadat přímo, v parametru MPI nazvaném `-hosts` lze však zadat soubor se specifikací použitých uzlů a počtu procesorů. Podle dostupné dokumentace by SGE mělo tento parametr respektovat a úlohu spustit podle požadavků. To se však v našem případě nestalo. Museli jsme tedy úlohy spouštět ručně bez dávkového systému, takzvaně interaktivně, a přitom kontrolovat, aby na některém z uzlů nebylo spuštěno příliš mnoho úloh.




*Graf 9: Vliv počtu alokovaných procesorových jader na uzlech Dell*




*Graf 10: Vliv počtu alokovaných procesorových jader na uzlech Sun*

Graf hodnot naměřených pro uzly Dell téměř přesně odpovídá testům, které před námi provedli výše zmínění autoři pro odlišnou úlohu. Druhý graf sice obsahuje jisté rozdíly, celkově však můžeme říci, že jsme jejich měření potvrdili, samozřejmě s přihlédnutím k tomu, že měření byla prováděna na jiných datech. Důvody pro naměřené chování můžeme s největší pravděpodobností hledat v architektuře jednotlivých uzlů. U procesorů firmy Intel na uzlech Dell je použit

symetrický multiprocesing (SMP), konkrétně jsou procesory spojeny se sdílenou pamětí prostřednictvím severního můstku (northbridge) čipové sady. Paměť je zde zřejmě při plném zatížení (a tím pádem zvýšené komunikaci s ní) úzkým hrdlem. Oproti tomu jsou procesory AMD a uzly firmy Sun příkladem architektury NUMA: každý procesor zde může velmi rychle přistupovat k jemu vyhrazené paměti prostřednictvím integrovaného paměťového řadiče a vysokorychlostní sběrnice. Ukázalo se, že toto řešení může vést k lepší škálovatelnosti paralelního systému.

# 6 Flow123d

Na Fakultě mechatroniky, informatiky a mezioborových studií Technické univerzity v Liberci je vyvíjen program s názvem Flow123d[6]. Jedná se o simulátor proudění a transportu rozpuštěných látek v rozpukaném porézním (horninovém) prostředí. V této kapitole si popíšeme základní vlastnosti programu, jeho vstup a výstup a podíváme se na některé jeho problémy a jejich možná řešení.

Vývoj vlastního simulačního nástroje vychází z poznatků geologů a hydrogeologů o charakteru puklinového prostředí a proudění v něm, které by se daly shrnout do následujících bodů:

- Horninu samotnou je možno považovat za zcela nepropustnou, resp. její propustnost je zanedbatelná vzhledem k propustnosti puklin.
- I v horninových tělesech, která řadíme mezi nejkompaktnější, existují četné pukliny tvořící puklinovou síť.
- Většina puklin jsou tzv. malé pukliny, jejichž charakteristická délka zpravidla nepřesahuje jeden metr.
- Proudění podzemních vod malými puklinami je velmi pomalé.
- Malé pukliny mají díky svému počtu značný celkový objem a tím hrají významnou roli v transportních procesech.
- Současnými prostředky je prakticky nemožné změřit všechny parametry důležité pro modely proudění a transportu pro všechny malé pukliny vyskytující se v dané oblasti. Malé pukliny proto není možno charakterizovat jinak než statisticky.
- Většina kapaliny je vedena prostřednictvím relativně malého počtu puklin, které mají velké rozměry a velkou propustnost.
- Nejrychlejší tok kapaliny je možno pozorovat na průsečnicích hydraulicky významných puklin.

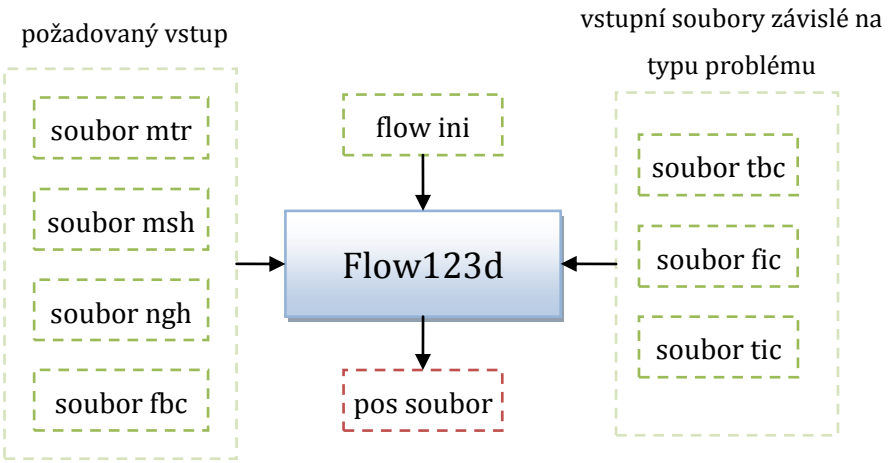
[6] Program je šířen pod licencí GNU GPL 3; stránky projektu: http://dev.nti.tul.cz/trac/flow123d

Vyvíjený model proudění kapaliny prostředím popisuje procesy ve třírozměrném prostředí (porézní bloky nahrazující pukliny malých rozměrů), dvourozměrných hydraulicky významných puklinách a jednorozměrných průsečnicích puklin a výměnu mezi jednotlivými dimenzemi. Výpočetní síť je z tohoto důvodu složena z třírozměrných čtyřstěnů, dvourozměrných trojúhelníků a jednorozměrných úseček.

Program je napsaný v jazyce C++ a je, jak již bylo napsáno, určen především pro výpočty na paralelních počítačích a tomuto se budou ostatně věnovat i všechny naše testy popsané v následující kapitole. Nasazení na paralelním počítači je možné díky knihovnám PETSc, které Flow123d využívá pro matematické výpočty, potažmo díky rozhraní MPI, na kterém PETSc staví a které se na několika místech přímo využívá i v tomto programu. Zároveň je však třeba si uvědomit, že zdaleka ne všechny kroky programu jsou v současné verzi prováděny paralelně a například načítaní sítě a dat, rozdělení úlohy na jednotlivé procesory a výstup programu je pouze sekvenční.

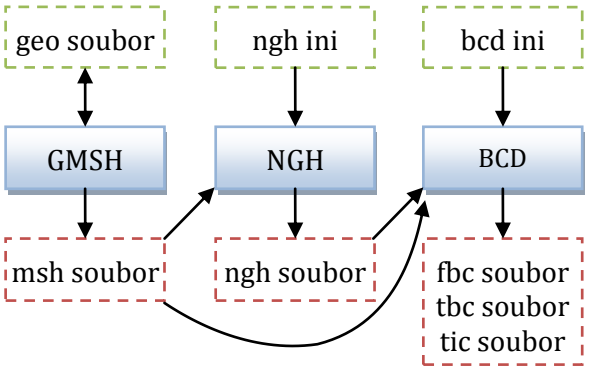
## 6.1 Spouštění, vstup a výstup programu

Spustitelný soubor přebírá důležitý parametr `-s`, kterým lze specifikovat vstupní soubor. Teprve tento textový `*.ini` soubor definuje vlastní vstupní soubory, konkrétně pak jde o soubor s modelovanou sítí (`*.msh`), popis materiálů (`*.mtr`) a specifikaci okrajových podmínek (`*.bcd`). Pro simulaci transportu se zde nachází ještě umístění souboru okrajových podmínek transportu (`*.tbc`) a počátečních podmínek (`*.tic`). Tuto situaci postihuje obrázek 6.




*Obr. 6: Schéma výpočtu*

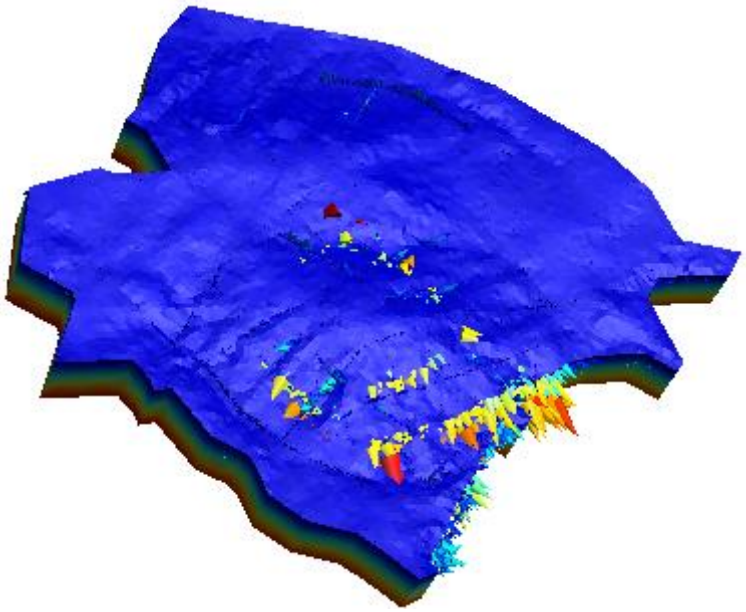
Pro přípravu výše popsaných vstupních souborů můžeme použít několik nástrojů. Obvykle je výchozím bodem soubor `*.geo` s popisem geometrie, který slouží jako vstup programu GMESH[7], pomocí kterého se vytvoří soubor definující síť. Dalším krokem je pak příprava souborů pro popis sousedností a okrajových podmínek, a to pomocí programů ngh a bcd (oba dva nástroje jsou distribuovány spolu s Flow123d).




*Obr. 7: Příprava vstupních souborů*

[7] Pod licencí GNU GPL dostupný na stránkách http://www.geuz.org/gmsh/

Samotným výstupem programu Flow123d je buď `*.pos` soubor, pro jehož vizualizaci lze použít již zmíněný program GMESH, nebo lze výstup zapsat ve formátu VTK, který je akceptován programem Paraview[8]. Příklad vizualizace výstupu z programu GMESH je uveden na obrázku 8.



*Obr. 8: Příklad vizualizace výstupu*

## 6.2 Profiler

Jedním z nedostatků programu Flow123d byl nepřívětivý způsob měření časových intervalů (doby provedení určitého úseku kódu) spolu s tím, že nebylo možné oddělit získané informace o naměřených časech od výstupu programu, a tak byly tyto informace „rozesety“ ve výstupním souboru a musely se dodatečně dohledávat (například pomocí regulárních výrazů). Kód byl především náchylný k chybám (nesměli jsme zapomenout na „úklid“ objektů) a ani z objektově orientovaného pohledu nebyl příliš dobře navržen – jednalo se spíše o dočasné řešení.

[8] Open-source nástroj pro vizualizace; stránka projektu http://www.paraview.org/

### 6.2.1 Návrh nového profileru

V odstavci výše jsme si představili důvody, které vedly k vytvoření nového nástroje pro měření časových intervalů – Profileru. Tento nástroj, reprezentovaný ve zdrojovém kódu několika třídami (které budou níže popsány), by měl splňovat především tři následující požadavky:

- Minimalizovat chyby programátora a možné úniky paměti při zachovaní snadné použitelnosti
- Z naměřených hodnot vypočítat základní charakteristiky (minimum / maximum času mezi jednotlivými procesory, díky čemuž můžeme určit, jak dobře je úloha rozložena mezi procesory, a podobně)
- Umožnit formátovaný výstup do zvláštního souboru s přidáním některých doplňujících informací, jako je velikost úlohy, datum a čas začátku a konce běhu a podobně

Základní předpoklad, na kterém je založeno měření časových intervalů a ostatně i návrh celého profileru, spočívá v myšlence, že se programátor nemusí starat o „úklid“, tedy explicitně specifikovat, kdy chce měření času ukončit – to bude automaticky ukončeno na konci programového bloku, ve kterém bylo spuštěno, lze ho však samozřejmě ukončit i dříve. Příklad tohoto přístupu je zachycen na níže uvedené ukázce zdrojového kódu. Tak jako mohou být do sebe vnořovány bloky programu (cykly a smyčky do funkcí, funkce může být volána jinou funkcí), tak lze do sebe vnořovat časové úseky, které ve výsledku tvoří hierarchickou stromovou strukturu – kořen stromu je vytvořen automaticky a představuje tím pádem dobu běhu celého programu.

Výše uvedené časové úseky jsou v návrhu reprezentované třídou `Timer`, a proto jim dále budeme říkat časovače. Centrálním bodem profileru je však třída `Profiler`, která má na starosti nejdůležitější operace jako spouštění časovačů či výstup do souboru. Kromě těchto dvou tříd je zde implementována ještě pomocná třída `TimerFrame`, se kterou však pravděpodobně nepřijdeme přímo do styku. Všechny uvedené třídy si podrobněji popíšeme dále.

```
START_TIMER("PREALLOCATION");

if (schur0 == NULL) { // create Linear System for MH matrix
     xprintf( Msg, "Allocating MH matrix for water model ... \n ");

     ...

}

END_TIMER("PREALLOCATION");
START_TIMER("ASSEMBLY");

xprintf( Msg, "Assembling MH matrix for water model ... \n " );

...
```

*Ukázka 1: Měření časových intervalů s novým profilerem*

#### 6.2.1.1 Měření pomocí maker

Jakým způsobem zařídit, aby se časovač ukončil ve chvíli, kdy program dorazí na konec programového bloku, ve kterém byl časovač spuštěn? Odpověď na tuto otázku souvisí i se způsobem, jakým se s časovači pracuje – v drtivé většině případů programátor využije možnosti spustit časovač prostřednictvím makra `START_TIMER`. Toto makro interně vytváří proměnnou typu `TimerFrame`, která teprve spouští vlastní časovač, a která bude na konci programového bloku odstraněna a zavolán její destruktor. Toto je velmi důležitý bod – pokud bychom místo „obyčejné“ proměnné vytvořili instanci objektu pomocí klíčového slova `new`, získali bychom pouze ukazatel na nový objekt a nedošlo by k jeho automatickému odstranění na konci bloku programu. Třída `TimerFrame` je pouze pomocná, používá se výhradně pro toto automatické ukončování časovačů a programátor s ní pravděpodobně vůbec nepřijde do styku. Vedle makra `START_TIMER` je k dispozici i makro `END_TIMER`, které slouží k explicitnímu ukončení časovače (před tím, než vykonávání kódu postoupí na konec programového bloku, kde by se časovač ukončil automaticky). Použití těchto maker je znázorněno na jednoduché ukázce kódu uvedené výše.

Až po tuto chvíli je vše v pořádku a bezproblémové, komplikace však nastávají, pokud si uvědomíme, že po použití makra `END_TIMER` můžeme opět zavolat `START_TIMER`. To, jak víme z předchozího odstavce, vytváří pomocnou proměnnou

a druhé použití tohoto makra by tudíž způsobilo konflikt v názvu proměnných (jedna proměnná s daným názvem už byla definována prvním makrem, zároveň však druhé makro nemá žádný nástroj, jak zjistit, že daná proměnná již existuje). Musíme tedy zajistit, že názvy proměnných vytvářených makrem `START_TIMER` budou (v rámci jednoho souboru) unikátní. K tomu nám dobře poslouží direktiva preprocesoru `__LINE__`, umožňující na její místo vložit v době překladu číslo aktuálního řádku, musíme však předpokládat, že nepoužijeme makro `START_TIMER` dvakrát na jednom řádku. Nyní nám tedy toto makro vytváří proměnné s unikátními názvy, avšak díky tomu makro `END_TIMER` neví nic o názvech použitých proměnných. Je tedy potřeba nějakým způsobem uchovávat informace o těchto názvech – v našem případě je toto realizováno asociativním polem, objektem typu `map`, jehož klíčem je textový popisek (tag) a hodnotou pak příslušející instance třídy `TimerFrame`. Při ukončení časovače pomocí makra v této datové struktuře vyhledáme příslušející instanci a ta teprve ve své metodě `close()` volá metodu profileru pro ukončení časovače. Aby ukončování fungovalo i v situacích, kdy programátor použije makro `START_TIMER` se stejným tagem vícekrát po sobě a následně také volá odpovídající počet makra `END_TIMER` (taková situace nemá žádný praktický význam, je však zapotřebí, aby měření časových intervalů dávalo správné výsledky), obsahuje objekt typu `TimerFrame` odkaz na nadřazený objekt se stejným tagem (odkaz na proměnnou deklarovanou startovním makrem). Při ukončování se hodnota v asociativním poli nahradí tímto nadřazeným objektem a je tak zaručeno, že se bude vždy pracovat se správnou instancí.

#### 6.2.1.2 Třída Profiler

Až doposud jsme se věnovali pomocné třídě `TimerFrame`, která je sice důležitá a nesmírně zjednodušuje používání profileru, nicméně nepředstavuje jeho ústřední část. Za „srdce“ profileru by se dala označit třída `Profiler` – ta implementuje návrhový vzor singleton, díky čemuž může existovat pouze jediná instance této třídy. To je v našem případě realizováno privátním konstruktorem (proto, aby nikdo nemohl svébytně vytvářet instance) a k profileru se přistupuje skrze statickou metodu `instance()` – ta vrací ukazatel na jedinou existující instanci (a případně

tuto instanci vytváří, pokud žádná neexistuje). Tímto způsobem mohou pomocné objekty typu `TimerFrame` (popisované v předchozím odstavci) pohodlně komunikovat s profilerem, čehož bychom sice stejně tak mohli dosáhnout i globální proměnnou, nebylo by to však tak elegantní jako toto řešení a navíc bychom se museli spoléhat na to, že globální proměnná již byla dříve správně inicializována. V našem případě není profiler inicializován při prvním volání metody `instance()`, ale explicitně metodou `initialize()`, která ve svém parametru předává MPI komunikátor, pomocí něhož budeme při tvorbě výstupního souboru „sbírat" výsledky z jednotlivých procesorů, na kterých program běžel. Výstup profileru je zapsán do souboru v okamžiku, kdy je jeho instance likvidována, tedy v destruktoru. To však v některých situacích může být příliš pozdě nebo chceme mít kontrolu i nad ukončením profileru, a proto je zde k dispozici metoda `uninitialize()`.

Samotná třída `Profiler` nám nenabízí nijak vysoký počet metod – těch veřejných je pouze sedm, a to včetně tří statických, o kterých jsme se již zmínili. Dvěma dalšími metodami, víceméně pouze pomocnými pro nastavení doplňkových informací, které budou vypsány do výstupního souboru, jsou `set_timer_subframes()` specifikující počet podúloh časovače (typicky se jedná o počet iterací) a dále pak `set_task_size()` pro nastavení velkosti řešeného problému. Tyto funkce používáme z toho důvodu, že je vcelku užitečné mít ve výstupním souboru výše uvedené informace, nicméně profiler nemá žádnou možnost, jak si informace sám obstarat, musíme mu je tedy předat tímto způsobem.

Názvy zbývajících dvou metod – `start()` a `end()` přesně vyjadřují jejich účel a daly by se označit za nejdůležitější metody profileru. Jak jsme si pověděli na začátku kapitoly, časovače tvoří hierarchickou strukturu a právě její vytváření mají na starosti tyto dvě metody. Aby výsledná struktura byla stromem, spouštíme automaticky po vytvoření profileru hlavní „kořenový" časovač, čímž bez jakékoliv námahy získáme dobu běhu celého programu. Stromovou strukturu tvoříme prostým způsobem: uchováváme si referenci na aktuální časovač a při spuštění nového časovače ho přidáme do seznamu potomků a prohlásíme ho za aktivní. Naopak při ukončení časovače ukončíme také všechny jeho potomky a jeho rodič se

stane aktivním. Je dobré upozornit, že nemůže existovat více časovačů se stejným popiskem (tagem) a pokud již časovač někde ve stromu existuje, nelze ho přemístit. Pokud tedy spouštíme časovače se stejnými tagy z různých míst, nemusí výsledná stromová struktura přesně odpovídat zanoření spouštěcích a ukončovacích maker, to však nemá na funkci profileru žádný vliv a ovlivní to pouze odsazení či zanoření časovačů ve výstupním souboru.

#### 6.2.1.3 Třída Timer

Instance třídy `Timer` představující objekty časovačů jsou poměrně jednoduché a slouží především k ukládání hodnot o času, počtu spuštění časovače, počtu podúloh nebo k uchování stromové struktury – k pohodlné práci je tu potřeba ukazatel na nadřazený časovač a dále pak seznam podřízených časovačů. To, co pravděpodobně stojí za zmínku, je způsob ukončování časovačů, i když jeho princip je také jednoduchý: v normálním případě se měření času ukončí až v situaci, kdy zavoláme metodu pro ukončení tolikrát, kolikrát metodu pro start (obvykle je však voláme pouze jednou) a zároveň při ukončení jsou automaticky ukončeny i všechny podřízené časovače.

#### 6.2.1.4 Výstup

Jak jsme již zmínili, výstup profileru probíhá do separátního textového souboru, jehož název začíná řetězcem „profiler_“, za nímž následuje čas ve formátu *den-měsíc-rok_hodina:minuta:sekunda*, čímž dosáhneme jisté jedinečnosti názvu a můžeme tak výstup z několika běhů programu provádět do stejného adresáře, aniž by se výstupní soubory přepisovaly (za předpokladu, že nebyly spuštěny přesně ve stejný čas, v tom případě by názvy souborů kolidovaly). Čas v názvu souboru představuje okamžik spuštění profileru, což má i jeden ryze praktický důsledek – pokud se v souborovém manažeru podíváme na adresář s výstupním souborem, tak podle času vytvoření (odpovídá času ukončení profileru) a názvu souboru můžeme na první pohled alespoň přibližně určit dobu běhu úlohy.

Ve výstupu jsou přehledně zapsány všechny důležité informace, jako je počet procesorů, na kterých úloha běžela, velikost úlohy nebo čas začátku a konce běhu. Dále zde nalezneme informace o časovačích, jejichž stromová struktura je

znázorněna odsazením, s jejich hodnotami uvedenými v jednotlivých sloupcích. Sloupec první udává počet volání časovače a v případě, že se počet na některých procesorech liší, je za číslem uvedena hvězdička. Ve druhém sloupci je uveden naměřený čas časovače, resp. aritmetický průměr času ze všech procesorů a v dalším sloupci následuje poměr nejkratšího a nejdelšího času mezi procesory. Tento poměr by se ideálně měl blížit 1, neboť zjednodušeně řečeno vyjadřuje, jak dobře je úloha mezi procesory rozdělena. Zmíněné hodnoty získáme z jednotlivých procesorů prostřednictvím rozhraní MPI, a to za pomoci takzvaných redukčních operací, konkrétně pak voláním funkce `MPI_Reduce`().

```
No. of processors: 12
Task size: 23684
Start time: 05/16/11 20:49:25
End time: 05/16/11 20:49:40
--------------------------------------------------
                            calls time  min/max subframes
WHOLE PROGRAM               1     15.38 0.98
 PREALLOCATION              1     0.02  0.25
 ASSEMBLY                   1     0.04  0.40
 SOLVING MH SYSTEM          1     0.68  0.79    40
  Schur 1                   1     0.12  0.85
  Schur 2                   1     0.07  0.86
 TRANSPORT                  1     0.38  0.76
  transport_matrix_assembly 1     0.01  0.00
   matrix_assembly_mpi      1     0.01  0.00
  transport_steps           1     0.36  0.74
   transport_step           578   0.19  0.76
```

*Ukázka 2: Příklad výstupního souboru*

## 6.3 Skripty pro automatické testování

Abychom zjistili, jak dobře si program Flow123d vede při výpočtu různých úloh, musíme provést sadu testů a vyhodnotit jejich výsledky. Již v páté kapitole při testování úloh z knihoven PETSc jsme zjistili, že i samotné spouštění úloh může být poměrně pracné. To je způsobeno jak samotným počtem úloh, jelikož provádíme více měření, abychom eliminovali odchylky, tak především nutností vytvářet ručně množství spouštěcích `qsub` souborů. To je případ pouze superpočítače Rex, u něhož musíme specifikovat počet procesorů ve spouštěcím souboru. Pro otestování jedné úlohy tedy potřebujeme vytvořit například deset souborů, lišících se od sebe pouze

tím, kolik procesorů je v nich specifikováno, a následně je všechny předat dávkovému systému. U clusteru Hydra je situace o něco jednodušší – jelikož se počet procesorů specifikuje jako parametr v příkazovém řádku, postačí nám jediný spouštěcí soubor. Tím však není všechna práce hotova. Výstup programu je třeba zpracovat, to znamená projít vytvořené výstupní soubory, nalézt v nich požadované informace a ty dát do vzájemných souvislostí, popřípadě z nich vypočítat další statistiky, jako například efektivitu. Za tímto účelem jsme navrhli několik skriptů, které jsou schopny daný problém do velké míry zautomatizovat.

### *6.3.1 Výběr programovacího jazyka*

Důvody, proč jsme se rozhodli pro některý ze skriptovacích jazyků a ne pro „klasické“ kompilované, jsou vcelku zřejmé: rychlost ani paměťová náročnost nejsou pro nás kritické, vše co potřebujeme, je z předaných parametrů vytvořit spouštěcí soubor, úlohu spustit a po jejím ukončení zpracovat její výstup. Při rozhodování, jaký konkrétní skriptovací jazyk použít, padla volba na jazyk Python. Je spíše otázka osobních preferencí nežli nějakého objektivního porovnání, který z jazyků je pro danou úlohu vhodný. Obecně lze říci, že bychom mohli použít v podstatě libovolný, i když například v interpretu bash je práce s desetinnými čísly dosti komplikovaná. Python má tu výhodu, že je (například oproti jazyku Perl) snadno čitelný, jelikož samotná specifikace jazyka definuje zásady pro lepší čitelnost, a má proto vysokou produktivitu psaní programů. Jeho objektově orientovaný přístup je pak již jen pomyslnou třešničkou na dortu, nicméně v našich skriptech nehraje významnou roli. Navíc je jeho běhové prostředí nainstalováno na obou testovacích počítačích, jeho použití tak nic nebrání.

### *6.3.2 Návrh skriptů*

Počáteční návrh skriptů vycházel samozřejmě z požadavků na zjednodušení práce tak, jak jsme si je popsali výše, nicméně již během návrhu a prvotní implementace vyplynuly na povrch další požadavky, které by se ve výsledku daly shrnout do následujících bodů, popisujících základní koncept a princip toho, jak skripty fungují:

- Pro spuštění testů musí uživatel zadat pouze minimum informací. Konkrétně si vystačíme pouze s jedním parametrem specifikujícím adresář se vstupními soubory.
- Úlohu spouštíme na různém počtu procesorů, čímž vlastně provádíme test zrychlení, a na každém počtu procesorů úlohu spustíme několikrát, abychom eliminovali odchylky vzniklé tím, na jaké uzly dávkový systém úlohu rozdělil. Počty procesorů, stejně tak jako počet běhů na každém procesoru, jsou přehledně nastaveny přímo ve zdrojovém kódu skriptu.
- Vstupní soubory jsou pro každý běh úlohy zkopírovány do zvláštního adresáře, což sice zvyšuje nároky na diskovou kapacitu, nicméně tím zajistíme, že se jednotlivé úlohy nebudou navzájem ovlivňovat a například si přepisovat výstupní soubory.
- Pro každý běh úlohy vytvoříme spouštěcí soubor a ten následně odešleme jako požadavek na zpracování dávkovému systému.
- Počkáme na dokončení běhu všech úloh, jednotlivé výstupní soubory profileru zpracujeme a vytvoříme z nich souhrnný přehled. Pro data z jednotlivých časovačů spočítáme efektivitu.

### *6.3.3 Implementační detaily*

Z předchozího odstavce víme, co by skripty měly dělat, a zbývá nám tedy krátce se seznámit s tím, jak konkrétně je požadovaná funkcionalita implementovaná. Zatím jsme intuitivně hovořili o skriptech v množném čísle a opravdu se v našem případě nejedná o skript jediný, ale celkem o tři.

#### 6.3.3.1 testrun.py

První skript je velmi jednoduchý. Ačkoli se v podstatě jedná o hlavní skript, se kterým bude uživatel pracovat, jediné co dělá, je kontrola jediného vstupního parametru (test, zda předaný adresář existuje) a spuštění dalšího skriptu na pozadí. Právě spuštění na pozadí je důležité (a je hlavním důvodem, proč jsme vytvořili tento skript), jelikož další skript čeká na dokončení běžících úloh a uživatel by tak nemohl dále pokračovat v práci.

#### 6.3.3.2 run_back.py

Jak již název napovídá, jedná se o skript, který je určen k tomu, aby běžel na pozadí. Vstupem je opět pouze adresář se vstupními soubory a skript pracuje níže uvedeným způsobem.

Nejprve vytvoříme nový adresář, jehož název reprezentuje aktuální čas, čímž získáme unikátní hodnotu (pokud skript nevoláme vícekrát během jedné sekundy) a navíc má tato hodnota na první pohled nějakou vypovídací hodnotu. V něm pak vytvoříme adresář pro každý běh úlohy – standardně se úloha spouští na 1, 2, 4, 6, 8, 10, 12, 16, 20, 24 a 32 procesorech vždy třikrát, ve výsledku tedy vytvoříme 33 adresářů. Do nich následně překopírujeme obsah adresáře se vstupními soubory (je zde k dispozici proměnná nastavující, zda se mají kopírovat i vnořené adresáře). Při kopírování ignorujeme soubory `*.pos` (výstupní sobory z programu Flow123d, které mohou být poměrně velké) a rovněž hledáme vstupní `*.ini` soubor.

Poté, co jsou všechny potřebné soubory na svém místě, je potřeba úlohy spustit. K tomu využijeme skriptů, které paralelně během tvorby této diplomové práce vytvářel Michal Nekvasil [1]. Tyto skripty jsou psány pro interpret bash a na základě systému, na kterém běží (specifikováno jako parametr), vytváří spouštěcí `*.qsub` soubor, který následně odešlou dávkovému sytému ke zpracování. Kooperace s těmito skripty je zároveň jediným bodem v našich programech, jemuž musí uživatel při prvním použití na novém systému věnovat pozornost, jinak by se mohlo stát, že skripty budou vytvářet spouštěcí soubory ve špatném formátu a dávkový systém je nebude schopen spustit. V našem kódu jsou připraveny příkazy použitelné pro počítače Hydra i Rex s tím, že vždy jeden příkaz je zakomentován a není tak použit.

Použité skripty, o kterých jsme se zmínili v předchozím odstavci, při odeslání spouštěcího souboru dávkovému systému vytváří i dočasný soubor `lock`, který je po dokončení běhu úlohy odstraněn, čehož my můžeme využít pro kontrolu toho, zda všechny spuštěné úlohy již skončily. V našem případě každou minutu kontrolujeme všechny vytvořené adresáře, a pokud v nich nalezneme soubor `lock`, víme, že úloha stále ještě běží. Pokud již běh všech úloh skončil, spustíme nakonec

ještě skript pro tvorbu výsledného přehledu (reportu). Důvodem, proč automaticky vytváříme výsledný přehled, není nic jiného, než snaha uživateli co nejvíce ulehčit práci tak, aby tuto činnost nemusel dělat „ručně“. Nutnost čekat na dokončení běhu všech úloh je už jen logickým důsledkem tohoto požadavku.

#### 6.3.3.3 make_report.py

Poslední skript, který je součástí našeho testovacího prostředí, má na starosti tvorbu přehledu (reportu) z dat ve výstupních souborech profileru. Ačkoli se ve skutečnosti jedná o nejrozsáhlejší skript, jeho činnost je poměrně přímočará, a tak ani jeho popis zde nebude příliš zdlouhavý.

Tak jako dva předchozí skripty, i tento přebírá v jediném svém parametru cestu k adresáři, samozřejmě však ne k adresáři se vstupními soubory jako oba předchozí, ale k adresáři vytvořenému skriptem `run_back.py` s podadresáři pro každý jednotlivý běh úlohy. Základem je projít všechny tyto podadresáře a v každém nalézt výstupní soubor profileru. V něm pak pomocí regulárních výrazů nalezneme počet procesorů, velikost úlohy a informace o naměřených časech. Časovače, které chceme zpracovat, nastavujeme přehledně v podobě seznamu textových řetězců jako globální proměnnou na počátku skriptu. Aby bylo možno získané informace pohodlně porovnávat, převedeme je z textové podoby do číselné, a aby se s nimi i dobře pracovalo, uložíme je v podobě objektů.

Pokud máme k dispozici seznam objektů představujících informace o jednotlivých výstupních souborech profileru, zjistíme, zda se jedná o úlohu konstantní velikosti, která běžela na různém počtu procesorů a data tak představují test zrychlení, nebo zda jde o běhy různých velikostí jedné úlohy a jedná se tak o test škálovatelnosti. Se znalostí těchto informací pak již můžeme poměrně snadno vypočítat efektivitu pro každý běh a každý časovač a následně informace zapsat do souboru, pojmenovaného jednoduše `report`.

# 7 Testovací úlohy programu Flow123d

Při tvorbě programu, který je určen pro paralelní systémy (pro nás konkrétně Flow123d), není důležité pouze to, že program na systému funguje a produkuje správné výsledky – to považujeme za samozřejmost. Důležité je zde zejména to, jak se program chová pro různé úlohy, jejich velikosti a na různém počtu procesorů. Je pochopitelné, že výpočet velké úlohy na malém počtu procesorů může trvat dlouho – to je také důvod, proč výpočty provádět na superpočítači – je však zapotřebí, abychom při jeho použití získali nějaký benefit, to znamená kratší výpočetní časy při použití většího počtu procesorů a adekvátně tomu by nám použití většího počtu výpočetních jednotek mělo umožnit počítat větší úlohy ve stále přijatelném čase.

V této kapitole se zabýváme zkoumáním výše uvedených závislostí specifických pro program Flow123d[9] a několik úloh proudění či transportu na superpočítačích Hydra a Rex. Použité úlohy můžeme pro přehlednost rozdělit do dvou skupin – reálné, jejichž základem je nějaká skutečná lokalita, a čistě umělé (nabízí se i slovo akademické) úlohy, které se většinou zaměřují na otestování určitých funkcí či algoritmů v programu. V dalším textu si u jednotlivých úloh kromě samotných výsledků i podrobněji popíšeme, jaký je účel dané úlohy.

V kapitole 5 jsme při testech úloh z knihovny PETSc naměřený čas vztahovali k počtu iterací, jelikož nás při testech zrychlení a škálovatelnosti zajímalo především chování matematických operací. Zde nás bude povětšinou zajímat celkové chování úloh s tím, že důvody pro toto chování lze obvykle hledat právě v počtu iterací, a i o nich se tedy budeme často zmiňovat.

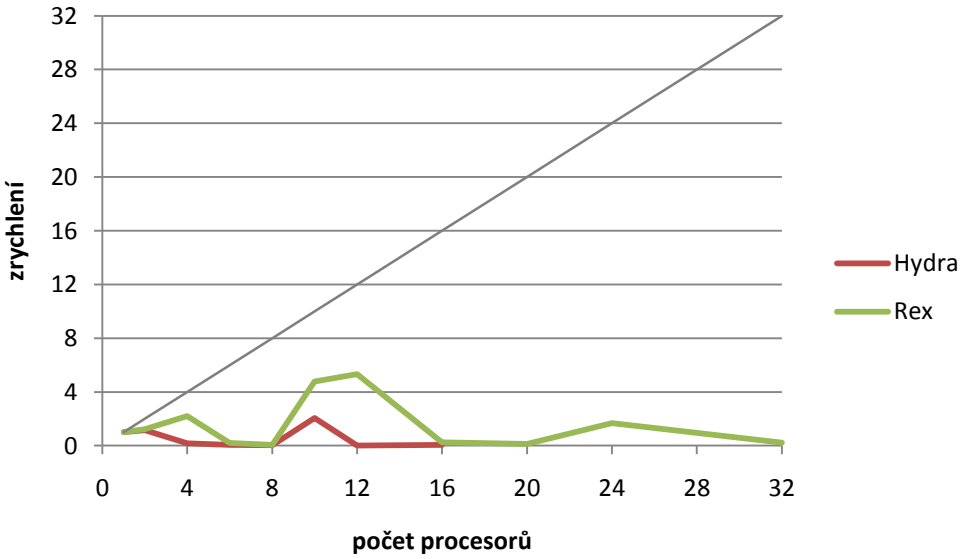
V páté kapitole jsme také zjistili, jaký má vliv použitý překladač – z tohoto důvodu jsou knihovny PETSc i program Flow123d na počítači Rex přeloženy překladačem icc, zatímco na clusteru Hydra je to kompilátor gcc. Bohužel v době testování byly některé z uzlů na clusteru Hydra v chybovém stavu a dávkový systém na nich úlohu nemohl spustit. Maximální počet procesorů, na kterém jsme testování na clusteru Hydra mohli provést, se tím pádem snížil na 16, což by za normálních

[9] Pro testování jsme použili vývojovou verzi 1.6.5

okolností byl dostatečný počet k tomu, abychom chování použitých úloh zjistili. Nicméně kvůli stavu, ve kterém se cluster během testování nacházel, se spousta úloh nepodařilo úspěšně dokončit, a to obvykle kvůli vynucenému ukončení programu (z nám neznámého důvodu) nebo kvůli nedostatku paměti. U některých měření jsou tak výsledky neúplné a i do některých grafů proto nemohly být hodnoty pro cluster Hydra vyneseny. Naměřené hodnoty, ze kterých jsme vycházeli a ze kterých také pocházejí uvedené grafy, jsou uvedeny v příloze.

## 7.1 Testovací úloha: Krychle

Tato úloha je ryze umělým testem zaměřeným pouze na proudění, transport se zde nepočítá. Základem je modelová síť tvořící krychli, ve které se nachází nejen dvourozměrné, ale i jednorozměrné pukliny. Tím je tato úloha specifická, žádná jiná totiž neobsahuje všechny tři typy elementů, od jednorozměrných až po třírozměrné. Abychom byli přesnější, jednorozměrná puklina se zde nachází pouze jedna a spojuje průsečíky úhlopříček dvou protilehlých stěn. Dvourozměrné pukliny se zde nacházejí dvě a jsou tvořeny na sebe kolmými rovinami procházejícími vrcholy krychle, jejichž průsečíkem je právě ona jednorozměrná puklina. Tato úloha je také vhodná pro test škálovatelnosti, jelikož použitá síť existuje v několika velikostech, které byly voleny právě s přihlédnutím k tomuto typu testu.




*Graf 11: Test zrychlení pro úlohu „Krychle“*

Grafy vytvořené z naměřených hodnot ukazují situaci, kterou jsme na začátku testu rozhodně nečekali. Je patrné, že na určitém počtu procesorů dojde k výraznému nárůstu doby běhu. Ten je způsoben extrémním zvýšením počtu iterací, a proto pokud bychom sledovali pouze efektivitu jedné iterace nebo počet matematických operací za sekundu (FLOPS), byly by výsledky relativně uspokojivé, nebylo by z nich však poznat, jak se úloha opravdu chová.

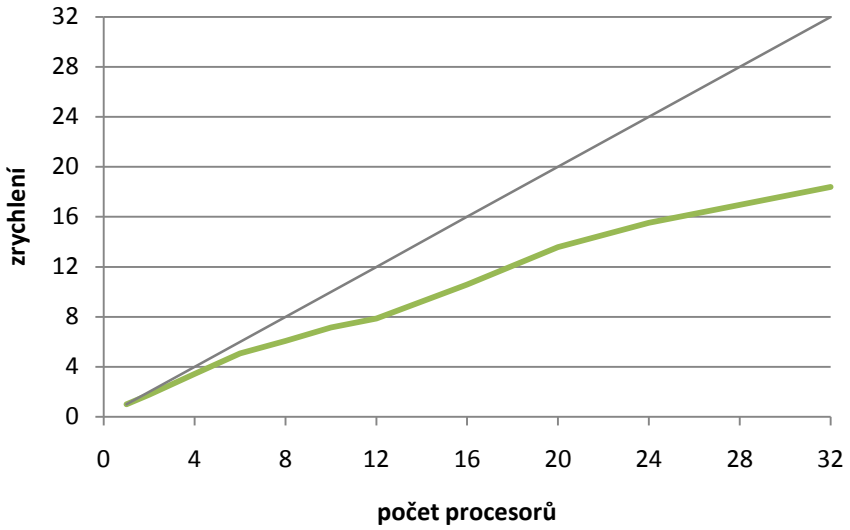
Co tedy vede k enormnímu zvýšení počtu iterací a tím pádem vyšším časům? Podle všeho je z části na vině samotný algoritmus rozkládající úlohu na jednotlivé procesory, přesněji řečeno knihovna METIS, která má tuto činnost na starosti. Vlastní důvod pak objasňuje hypotéza, že dojde k rozdělení jednorozměrné pukliny na více procesorů, což ovlivní použitý předpodmiňovač. Bohužel v době psaní práce nebyl k dispozici nástroj, který by nám dokázal zobrazit, jak je daná síť mezi procesory rozložena, a tak neexistuje přímý důkaz pro podpoření této teorie.

### *7.1.1 Volba předpodmiňovače*

Ačkoli nemůžeme zobrazit, jak je síť rozložena mezi procesory, můžeme se pokusit eliminovat vyšší počet iterací tím, že specifikujeme předpodmiňovač a jeho parametry, které má použít. Po vložení následujícího řádku do inicializačního `*.ini` souboru dojde k razantnímu zlepšení chování úlohy:

```
Solver_params = -ksp_type bcgs -ksp_diagonal_scale_fix -pc_type
asm -pc_asm_overlap 4 -sub_pc_type ilu -sub_pc_factor_levels 5
```

To, že skutečně dojde ke zlepšení chování, ukazuje graf 12. Situace je zajímavá tím, že se jedná o velmi podobnou metodu, kterou Flow123d standardně používá, avšak zde místo předpodmiňovače cg používáme bcgs, který nevyžaduje, aby byla matice pozitivně definitní (z tohoto důvodu používá Flow123d ještě parametr `sub_pc_factor_shift_positive_definite`, který by měl zajišťovat pozitivní definitnost matice).




*Graf 12: Test zrychlení pro úlohu „Krychle“, Rex*

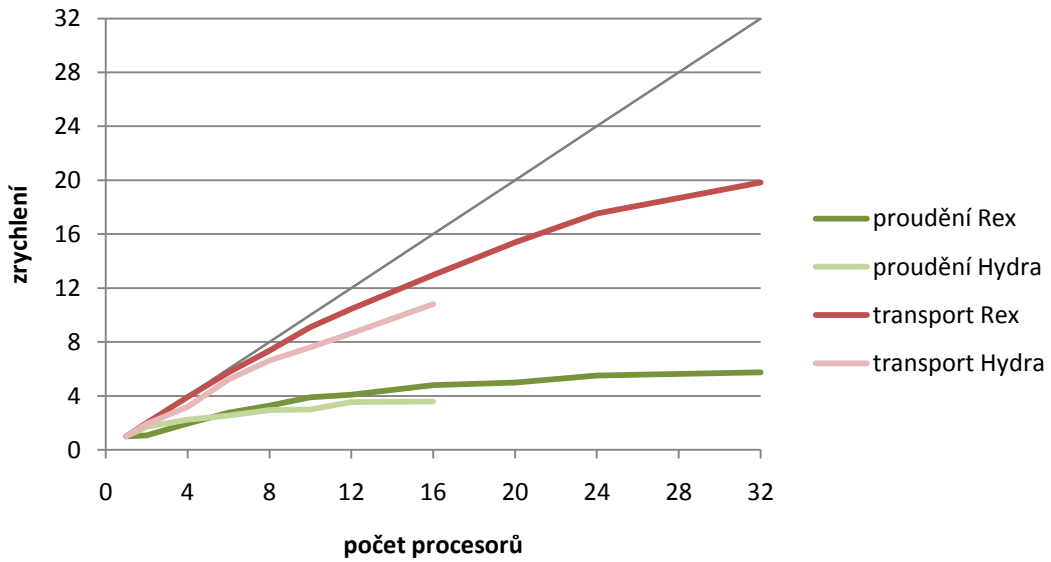
Je patrné, že použití nové metody opravdu výrazně zlepšilo chování úlohy – avšak pouze na počítači Rex, na clusteru Hydra se úspěšně dokončilo pouze velmi málo úloh. Těžko však za stavu, ve kterém se cluster Hydra v době testování nacházel, můžeme vinit použitý předpodmiňovač. Na počítači Rex sice došlo k mírnému prodloužení času výpočtů (v porovnání s časy původní úlohy a tam, kde se nevyskytoval problém s extrémním počtem iterací), nicméně počet iterací se víceméně ustálil. Ve světle nových výsledků jsme tedy mohli provést i test škálovatelnosti, jehož graf je uveden níže.


*Graf 13: Test škálovatelnosti pro úlohu „Krychle“, Rex*

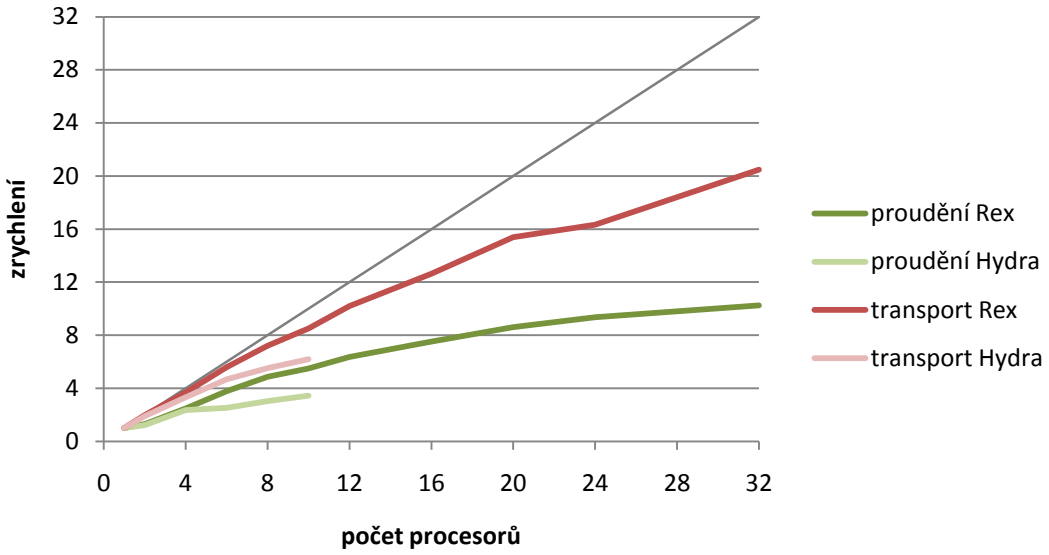
Naměřená efektivita při postupném zvětšování velikosti úlohy sice ze začátku poměrně strmě klesá, nicméně můžeme zde pozorovat tendenci ustálit se na vcelku přijatelné hodnotě, což by znamenalo pozitivní zprávu pro běh na větším počtu procesorů než námi testovaných 12.

## 7.2 Testovací úlohy: 2D proužek, 3D proužek a čtverec

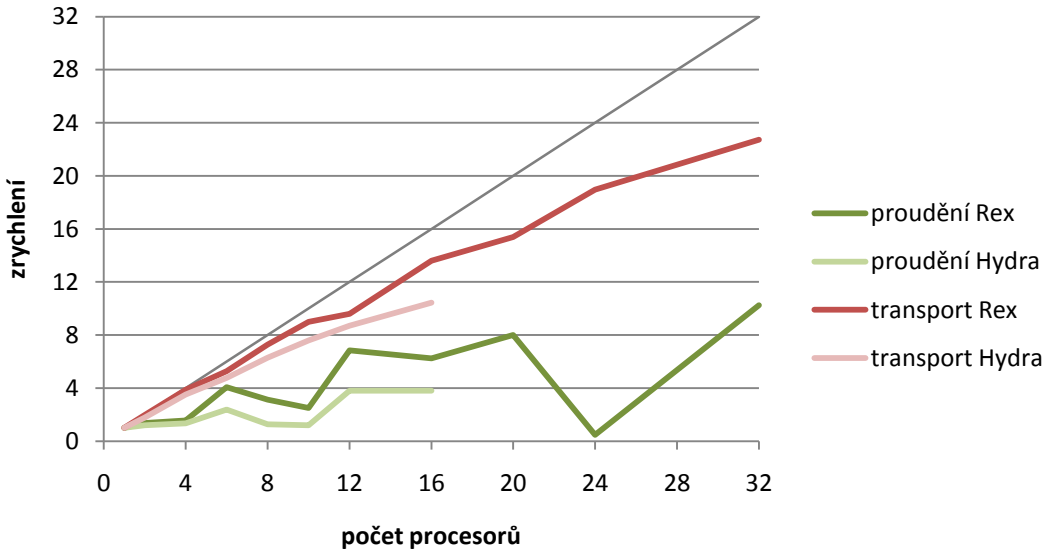
Tyto ryze umělé úlohy mají společné nejen to, že se u nich kromě proudění počítá i transport, ale především to, že byly vytvořeny až na základě neuspokojivých výsledků úlohy „Krychle“. Jedná se o úlohy, jejichž síť je velmi jednoduchá, bez jakýchkoli puklin a umožní nám tak sledovat chování programu v poměrně specifických situacích a zjistit tak, zda nepříznivé výsledky předchozího měření mají původ ve tvaru počítané sítě či její dimenzi. Síť dvourozměrného proužku má tvar velmi protáhlého obdélníku (poměr jeho stran je 1:100) a na jeho základě je vytvořena i síť pro 3D proužek – ta přestavuje kvádr s poměry stran 1:1:100. Všechny tři úlohy jsou k dispozici v několika velikostech, nicméně, jak se ukázalo, nemá velikost na efektivitu téměř žádný vliv, a tudíž zde budeme demonstrovat výsledky pouze na jedné velikosti od každé úlohy, a to konkrétně na následujících třech grafech:




*Graf 14: Test zrychlení pro 2D proužek*



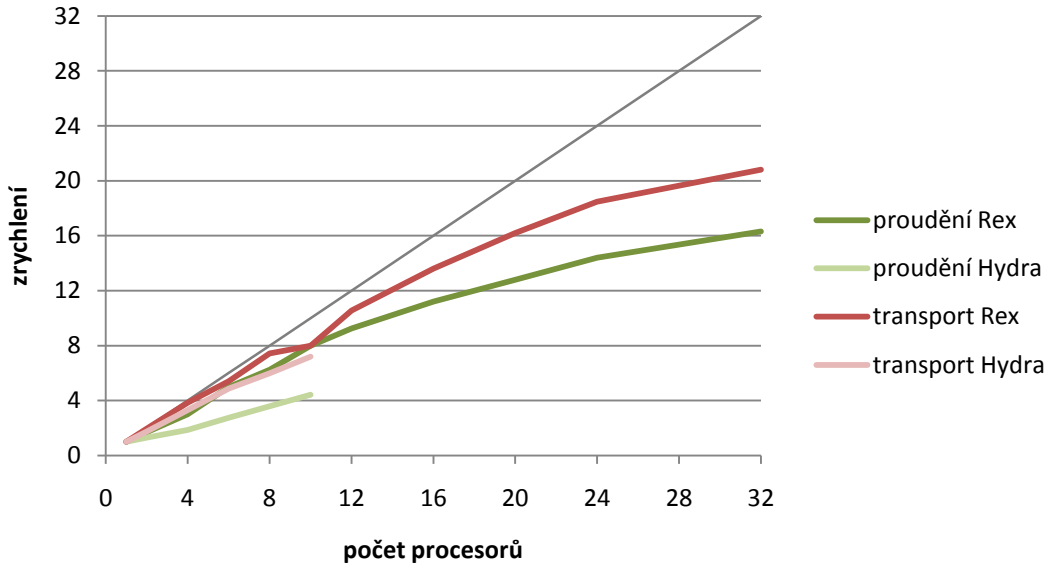

*Graf 15: Test zrychlení pro 3D proužek*




*Graf 16: Test zrychlení čtvercové sítě*

Test zrychlení čtvercové sítě vykazuje podobné rysy jako v případě krychle. Schválně jsme zde neuvažovali efektivitu vzhledem k počtu iterací, aby právě ono chování vyniklo – tak jako v případě krychle se zde pro určitý počet procesorů rapidně zvýší počet iterací a tím i celková doba, což se v grafu projeví ostrými zlomy. Tak jako v případě krychle můžeme specifikovat parametry předpodmiňovače, což náš problém vyřeší (viz graf 17). Je nicméně vhodné zamyslet se i nad důvody, proč se u čtvercové sítě tento problém vyskytuje a u obdélníkového proužku nikoliv. Zde bychom mohli opět uplatnit hypotézu o nevhodně rozdělené úloze mezi procesory a tím pádem vyšší komunikaci mezi nimi

– zatímco u obdélníku algoritmus pravděpodobně „chytře“ rozřeže síť krátkými řezy víceméně rovnoběžnými s kratší stranou, u čtverce, jehož žádný rozměr nepřevyšuje ostatní, mohou být na jeden procesor přiděleny i části sítě, které spolu vůbec nesousedí.

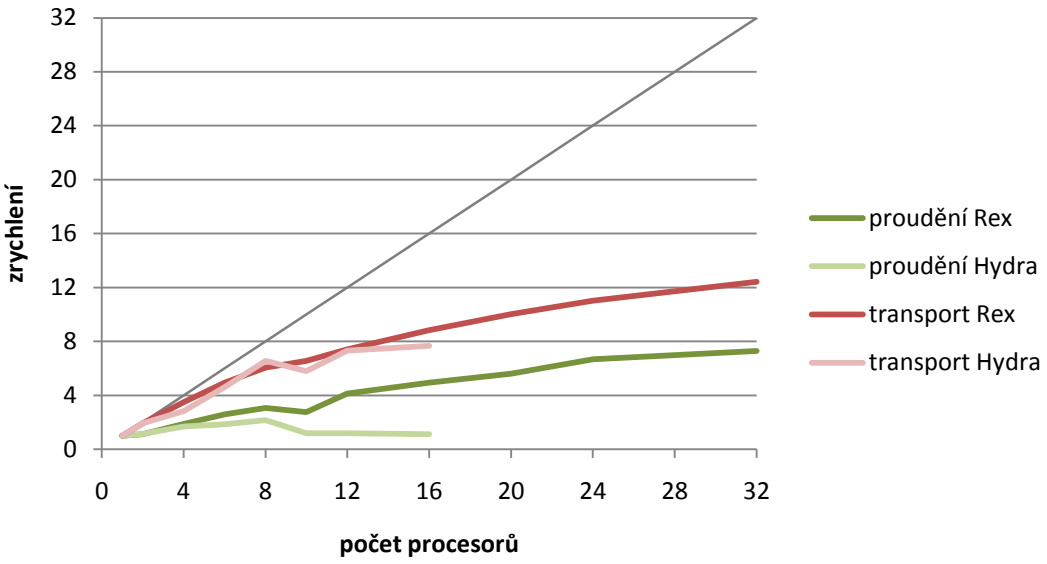



*Graf 17: Test zrychlení čtvercové sítě s upravenými parametry předpodmiňovače*

S novými parametry předpodmiňovače jsme eliminovali extrémní hodnoty iterací na některých počtech procesorů. Pokud nyní porovnáme grafy testů zrychlení pro dvourozměrný a třírozměrný proužek a čtverec s upravenými parametry předpodmiňovače, můžeme na všech pozorovat podobný průběh křivky pro transport. Díky tomu, že základním prvkem transportu je maticové násobení, je jeho efektivita lepší než efektivita řešiče proudění, což je chování, které jsme před vlastním započetím testů předpokládali. Výraznější rozdíly lze nalézt mezi efektivitou proudění u těchto třech úloh. Můžeme přitom porovnávat hodnoty efektivity 2D a 3D proužku s původním předpodmiňovačem a hodnoty čtverce s upravenými parametry, jelikož při novém nastavení předpodmiňovače se efektivita proudění 2D a 3D proužku změní pouze nepatrně. Z tohoto porovnání je zřejmé, že tvar sítě hraje v tomto případě daleko větší vliv než u transportu a efektivita je lepší u sítí, u kterých některý z rozměrů nepřevyšuje výrazným způsobem ty ostatní. Rovněž zde na počítači Rex opět vidíme vyšší efektivitu jak proudění, tak i transportu, což je jako v případě testů úloh z knihoven PETSc dáno jeho architekturou.

## 7.3 Testovací úloha: Melechov

V tomto případě se jedná o úlohu založenou na reálném základě, a to konkrétně na datech z oblasti Melechovského masivu ležícího zhruba 10 kilometrů severně od Humpolce. Vymezená oblast pokrývá území o rozloze zhruba 60 km$^2$. Síť této testovací úlohy obsahuje kromě třírozměrných elementů ještě dvourozměrné pukliny a existuje ve třech velikostech, z čehož dvě nejmenší jsou kromě výpočtů proudění připraveny i na transport. Pro přehlednost si zde uvedeme pouze graf pro úlohu střední velikosti, graf pro nejmenší úlohu je vcelku obdobný a zrychlení pro proudění na větší úloze se rovněž chová přibližně stejným způsobem.

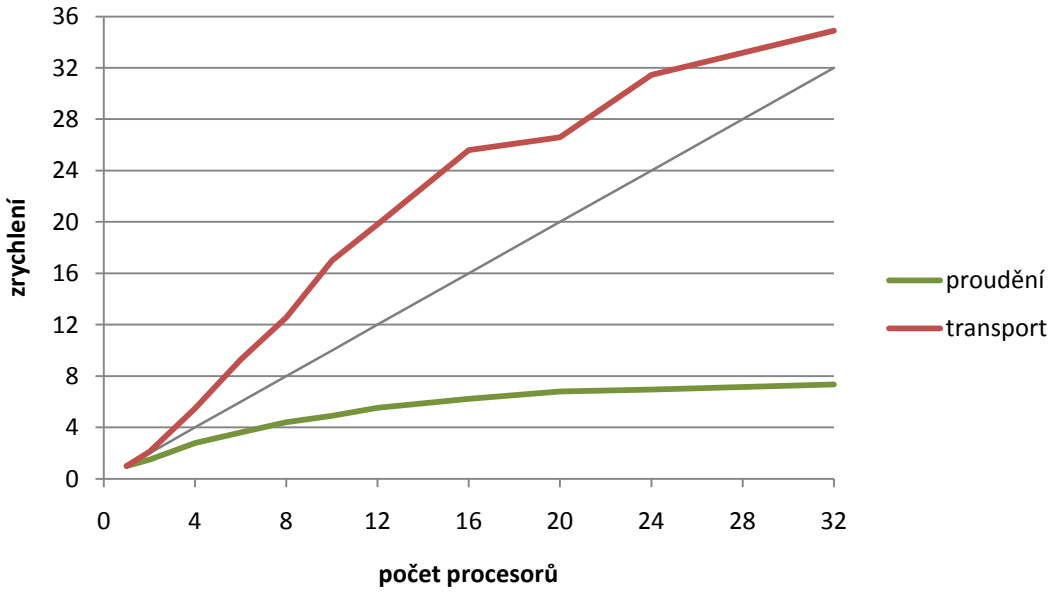



*Graf 18: Test zrychlení pro úlohu Melechov*

Oproti třem úlohám, které jsme si popsali výše, je zde efektivita transportu o něco horší. To však není nikterak překvapující, jelikož se jedná o nepoměrně složitější úlohu obsahující navíc i pukliny, a s přihlédnutím k tomuto můžeme efektivitu hodnotit stále jako velmi dobrou. Poněkud překvapivé je chování proudění na clusteru Hydra, u něhož při počtu deseti procesorů dojde k poklesu efektivity, není to však způsobeno zvýšením počtu iterací.

## 7.4 Testovací úloha: Decovalex

Další a zároveň poslední úloha má rovněž reálný podklad a byla vytvořena v rámci projektu Decovalex (DEvelopment of COupled models and their VALidation against EXperiment). Dvourozměrná síť je tvořena pouze jednorozměrnými puklinami (žádné jiné elementy se v ní kromě těchto nevyskytují) a jednotlivé její varianty byly vytvořeny z dat získaných působením tlaku z různých směrů na žulový blok. Podobně jako u první úlohy – krychle, i zde v `*.ini` souboru změníme parametry předpodmiňovače, a to naprosto totožně jako u první úlohy. Pokud bychom to neudělali, stejně jako v případě krychle a čtverce by byl běh úlohy v určitých okamžicích ukončen po dosažení 10000 iterací.




*Graf 19: Test zrychlení pro úlohu Decovalex na počítači Rex*

Opět máme k dispozici pouze výsledky ze superpočítače Rex a výsledky jsou to vskutku překvapivé – zrychlení je místy téměř dvakrát lepší, než je ideální stav. Nejedná se přitom o nějakou náhodu, podobné chování vykazovaly všechny tři varianty této úlohy. Důvod bychom pravděpodobně mohli hledat, stejně jako při testu první úlohy z knihoven PETSc, ve vyrovnávacích pamětech cache. Síť úlohy Decovalex není příliš rozsáhlá, a tím pádem je možno ji celou uložit v těchto rychlých pamětech. Maticové násobení, které je základem transportu, má pak z této situace očividně větší profit než výpočet proudění.

# 8 Závěr

V práci jsme si ukázali některé z možností, jakými mohou být vyvíjeny aplikace pro paralelní systémy, jmenovitě pak prostřednictvím rozhraní MPI a s použitím knihoven pro numerickou matematiku. Větší pozornost byla věnována zdarma dostupným knihovnám PETSc, na jejichž testovacích úlohách jsme mohli pozorovat rozdíly mezi dvěma použitými superpočítači, nazvanými Hydra a Rex. Odlišnosti jejich architektur vynikly při testech zrychlení a škálovatelnosti, mohli jsme také vidět podstatné zrychlení při použití překladače icc od firmy Intel, bohužel však pouze na počítači Rex. Rovněž jsme potvrdili, že na clusteru Hydra má na výkon vliv i způsob, jakým se úloze přidělují procesory na jednotlivých uzlech.

Následně jsme se věnovali simulátoru proudění podzemní vody Flow123d, u kterého jsme navrhli a implementovali nový způsob měření časových intervalů, který je jednodušší na používání a odstraňuje hlavní nedostatky původní metody. Dále byla pro základní automatizaci testů navrhnuta sada skriptů v jazyce Python, kterou jsme později použili pro usnadnění práce při spouštění vlastních testovacích úloh programu Flow123d. Tyto testy byly založeny jak na reálných, tak i na umělých datech a ukázaly, jak si testovaný program v různých situacích stojí. Prokázalo se v nich, že použité nastavení předpodmiňovače na některých úlohách (krychle, čtverec, Decovalex) způsobuje při určitém počtu procesorů enormní zvýšení iterací, což je chování, které sice dokážeme ovlivnit, je však s podivem, že na jiných úlohách, byť podobných (jako je 2D a 3D proužek), jsme nic podobného nepozorovali. Na ryze umělých úlohách jsme také zaznamenali velmi dobrou efektivitu transportu, která byla u reálné a složitější úlohy Melechov o něco horší, stále však uspokojivá, u druhé úlohy s reálným základem (Decovalex) dokonce díky pamětem cache a malé velikosti sítě o poznání lepší. Vysoký počet iterací v určitých situacích je tedy hlavní oblastí, které je potřeba při vývoji programu Flow123d věnovat pozornost. Největší problémy jsme však zaznamenali s univerzitním clusterem Hydra, jehož některé uzly se při testech programu Flow123d nacházely v chybovém stavu, což nám nejen radikálně omezilo počet procesorů, na kterých jsme mohli úlohy testovat, ale především spousta úloh nebyla vůbec korektně dokončena. Z tohoto důvodu jsme nemohli provést vzájemné porovnání obou počítačů pro všechny testovací úlohy.

# Seznam použité literatury


[1] NEKVASIL, Michal. *Systém automatického sestavování a testů pro simulátor proudění Flow123d*. Bakalářská práce. Fakulta mechatroniky, informatiky a mezioborových studií. Technická univerzita v Liberci. 2011.

[2] *PETSc Manual.* [online]. URL: <http://www.mcs.anl.gov/petsc/petsc-as>

[3] ŠEMBERA, Jan. *Mechanika tekutin – poznámky k teoretické části*. Fakulta mechatroniky, informatiky a mezioborových studií. Technická univerzita v Liberci. 2004. 63 s.

[4] ŠIDLOF, Petr, HORÁČEK, Jaromír, STANĚK, Jiří. *Parallel CFD simulation of flow on moving meshes on a heterogeneous computational cluster*.

[5] The University of Texas at Austin. *Batch Systems Comparison.* [online]. URL: <http://services.tacc.utexas.edu/index.php/batch-systems-comparison>

[6] TVRDÍK, Pavel. *Paralelní systémy a algoritmy*. Praha: ČVUT, 2000. 228 s. ISBN 80-01-02267-6

# Příloha A – specifikace superpočítačů Hydra a Rex

## Hydra

12 uzlů Dell PowerEdge 1950
- 2× Intel Xeon 5140 2.33GHz/4MB 1333FSB (2 jádra)
- 4GB RAM 667MHz
- 80GB SATA2, 7200 ot./min

17 uzlů Sun Fire V20z
- 2× AMD Opteron 252, 2600 MHz (1 jádro)
- 4 GB RAM
- 73 GB HDD, 10025 ot./min
- 1x Dual Ultra320 SCSI

propojovací síť 1 Gbps
OS: Linux CentOS 5.4
Rocks Clusters 5.3 - Rolled Tacos
adresa: hydra.kai.tul.cz

## Rex

16× jednojádrový Intel Itanium2, jádro Madison, 1.5GHz
80× dvoujádrový Intel Itanium2, jádro Montecito, 1.6GHz
celkem 270 GB RAM
diskové pole 1,2 TB
Sdílená paměť – architektura ccNUMA.
adresa: rex.civ.cvut.cz

# Příloha B – výsledky testů úloh z knihovny PETSc

Pokud není uvedeno jinak, představují hodnoty v tabulkách naměřený čas v sekundách, efektivita a zrychlení jsou stejně jako počet iterací bezrozměrné.

## Úloha ex3 – test zrychlení

| | | velikost: 200 | | | | | velikost: 500 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | *proc.* | *1.* | *2.* | *3.* | *Ø* | *efekt.* | *1.* | *2.* | *3.* | *Ø* | *efekt.* |
| Hydra | *1* | 267,5 | 274,2 | 251,8 | 264,5 | 1,0 | 14740,0 | 13560,0 | 14280,0 | 14193,3 | 1,0 |
| | *2* | 95,4 | 91,1 | 98,2 | 94,9 | 1,4 | 3852,0 | 3645,0 | 4023,0 | 3840,0 | 1,8 |
| | *4* | 23,3 | 25,3 | 23,5 | 24,0 | 2,8 | 1380,0 | 1302,0 | 1356,0 | 1346,0 | 2,6 |
| | *8* | 3,7 | 3,4 | 3,7 | 3,6 | 9,2 | 338,1 | 350,3 | 341,0 | 343,1 | 5,2 |
| | *16* | 1,3 | 1,4 | 1,2 | 1,2 | 12,8 | 77,0 | 76,2 | 78,2 | 77,1 | 11,5 |
| | *24* | 0,8 | 0,8 | 0,9 | 0,8 | 13,1 | 38,4 | 32,1 | 38,2 | 36,2 | 16,3 |
| | *32* | 0,5 | 0,6 | 0,5 | 0,5 | 15,8 | 26,5 | 26,9 | 25,8 | 26,4 | 16,8 |
| Rex | 1 | 315,5 | 324,4 | 224,9 | 288,2 | 1,0 | 15440,0 | 13560,0 | 11490,0 | 13499,0 | 1,0 |
| | 2 | 53,1 | 51,3 | 67,5 | 57,3 | 2,5 | 2879,0 | 2866,0 | 2807,0 | 2850,6 | 2,4 |
| | 4 | 12,9 | 21,0 | 17,9 | 17,3 | 4,2 | 827,4 | 962,7 | 1100,0 | 963,3 | 3,5 |
| | 8 | 3,1 | 3,1 | 3,5 | 3,2 | 11,0 | 266,3 | 170,8 | 267,5 | 234,8 | 7,2 |
| | 16 | 4,1 | 0,7 | 3,9 | 2,9 | 6,0 | 73,5 | 59,0 | 65,4 | 65,9 | 12,8 |
| | 24 | 3,91 | 6,5 | 0,7 | 3,7 | 3,2 | 37,5 | 32,2 | 37,6 | 35,8 | 15,7 |
| | 32 | 18,3 | 7,0 | 6,9 | 10,7 | 0,8 | 46,8 | 38,1 | 17,4 | 34,1 | 12,4 |

## Úloha ex12 – test zrychlení

| | *proc.* | *1* | *2* | *4* | *8* | *16* | *24* | *32* |
|---|---|---|---|---|---|---|---|---|
| Hydra | *1.* | 6963 | 3344 | 1967 | 1499 | 718,8 | 563,7 | 424,7 |
| | *2.* | 6997 | 3231 | 2831 | 1499 | 714,1 | 541,5 | 423,9 |
| | *3.* | 6444 | 3204 | 3167 | 1385 | 705,4 | 576,0 | 439,2 |
| | *Ø* | 6801,3 | 3259,67 | 2655,0 | 1461,0 | 712,7 | 560,4 | 429,2 |
| | *efekt.* | 1,0 | 1,0 | 0,6 | 0,5 | 0,5 | 0,5 | 0,4 |
| | *iterace* | 7550 | 6031 | 6617 | 6387 | 6004 | 6265 | 7361 |
| | *ef./iter.* | 1,00 | 0,83 | 0,56 | 0,49 | 0,47 | 0,42 | 0,48 |
| Rex | *1.* | 34010 | 13980 | 7495 | 3819 | 1841 | 1347 | 1186 |
| | *2.* | 35370 | 14280 | 7660 | 3813 | 1848 | 1378 | 1130 |
| | *3.* | 38370 | 14300 | 7513 | 3816 | 1827 | 1364 | 1159 |
| | *Ø* | 35916,67 | 14186,67 | 7556,00 | 3816,00 | 1838,67 | 1363,00 | 1158,33 |
| | *efekt.* | 1,0 | 1,3 | 1,2 | 1,2 | 1,2 | 1,1 | 1,0 |
| | *iterace* | 7550 | 6068 | 6221 | 6229 | 5999 | 6498 | 7362 |
| | *ef./iter.* | 1,00 | 1,02 | 0,98 | 0,97 | 0,97 | 0,94 | 0,94 |

## Úloha ex12 - test škálovatelnosti

| | proc. | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|---|
| | velikost úlohy | 500 | 707 | 1000 | 1414 | 2000 |
| Hydra | 1. | 41,7 | 135 | 163 | 298,6 | 697,7 |
| | 2. | 47,4 | 119 | 139,6 | 291,6 | 696,8 |
| | 3. | 47,4 | 93,9 | 167,6 | 341,9 | 696,5 |
| | Ø | 45,50 | 115,97 | 156,73 | 310,70 | 697,00 |
| | efektivita | 1,00 | 0,39 | 0,29 | 0,15 | 0,07 |
| | iterace | 859 | 1478 | 2211 | 2963 | 6004 |
| | efekt./iter. | 1,00 | 0,68 | 0,75 | 0,51 | 0,46 |
| Rex | 1. | 77,9 | 152,3 | 238,3 | 923,6 | 1602 |
| | 2. | 77,4 | 152,2 | 238,7 | 917,1 | 1835 |
| | 3. | 74,9 | 112,3 | 237,6 | 873,9 | 1918 |
| | Ø | 76,73 | 138,93 | 238,20 | 904,87 | 1785,00 |
| | efektivita | 1,00 | 0,55 | 0,32 | 0,08 | 0,04 |
| | iterace | 859 | 1478 | 2174 | 2935 | 6006 |
| | efekt./iter. | 1,00 | 0,95 | 0,82 | 0,29 | 0,30 |

## Úloha ex12 – porovnání uzlů výpočetního clusteru Hydra

| | proc. | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|---|
| Dell (Intel) | 1. | 7387 | 4319 | 2907 | 1500 | 702,3 |
| | 2. | 7450 | 4686 | 3192 | 1521 | 719,0 |
| | 3. | 7323 | 4434 | 2997 | 1527 | 701,6 |
| | Ø | 7386,7 | 4479,7 | 3032,0 | 1516,0 | 707,6 |
| | efektivita | 1,00 | 0,82 | 0,61 | 0,61 | 0,65 |
| | zrychlení | 1,00 | 1,65 | 2,44 | 4,87 | 10,44 |
| Sun (AMD) | 1. | 6258 | 3194 | 1974 | 1100 | 546,0 |
| | 2. | 6344 | 3287 | 2850 | 1104 | 545,3 |
| | 3. | 6265 | 2928 | 1645 | 1145 | 568,4 |
| | Ø | 6289,0 | 3136,3 | 2156,3 | 1116,3 | 553,2 |
| | efektivita | 1,00 | 1,00 | 0,73 | 0,70 | 0,71 |
| | zrychlení | 1,00 | 2,01 | 2,92 | 5,63 | 11,37 |

## Úloha ex12 – test vlivu překladače icc a knihovny MKL

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Hydra MKL | *proc.* | *1* | *2* | *4* | *8* | *16* | *24* | *32* |
| | *1.* | 8825 | 4716 | 3117 | 1498 | 713,1 | 478,5 | 432,8 |
| | *2.* | 8722 | 4601 | 3173 | 1394 | 713,5 | 480,9 | 432,8 |
| | *3.* | 8765 | 4615 | 3172 | 1499 | 716,4 | 480,5 | 421,8 |
| | *Ø* | 8770,7 | 4644,0 | 3154,0 | 1463,7 | 714,3 | 479,9 | 429,1 |
| | *efektivita* | 1,00 | 0,94 | 0,69 | 0,75 | 0,77 | 0,76 | 0,64 |
| | *iterace* | 7550 | 6031 | 6617 | 6387 | 6004 | 6265 | 7361 |
| | *ef. 1 iterace* | 1,00 | 0,75 | 0,61 | 0,63 | 0,61 | 0,63 | 0,62 |
| Rex MKL | *1.* | 33900 | 13870 | 7412 | 3695 | 1785 | 1190 | 1048 |
| | *2.* | 32930 | 13530 | 7276 | 3714 | 1795 | 1206 | 1040 |
| | *3.* | 33840 | 13440 | 7259 | 3668 | 1761 | 1195 | 1053 |
| | *Ø* | 33556,7 | 13613,3 | 7315,7 | 3692,3 | 1780,3 | 1197,0 | 1047,0 |
| | *efektivita* | 1,00 | 1,23 | 1,15 | 1,14 | 1,18 | 1,17 | 1,00 |
| | *iterace* | 7550 | 6068 | 6221 | 6229 | 5999 | 6498 | 7362 |
| | *ef. 1 iterace* | 1,00 | 0,99 | 0,94 | 0,94 | 0,94 | 1,01 | 0,98 |
| *Překladač icc* | *1.* | 12450 | 5494 | 3454 | 1456 | 788 | 556,2 | 444,5 |
| | *2.* | 12470 | 5498 | 3454 | 1467 | 798,7 | 533,2 | 451,1 |
| | *3.* | 12100 | 4705 | 3446 | 1465 | 802,5 | 536 | 445,5 |
| | *Ø* | 12340,0 | 5232,3 | 3451,3 | 1462,7 | 796,4 | 541,8 | 447,0 |
| | *efektivita* | 1,00 | 1,18 | 0,89 | 1,05 | 0,97 | 0,95 | 0,86 |
| | *iterace* | 7550 | 6009 | 6894 | 5547 | 6007 | 6198 | 7364 |
| | *ef. 1 iterace* | 1,00 | 0,94 | 0,82 | 0,77 | 0,77 | 0,78 | 0,84 |

# Příloha C – výsledky testovacích úloh programu Flow 123d

Tak jako v příloze B představují hodnoty v tabulkách naměřený čas v sekundách, efektivita a zrychlení jsou stejně jako počet iterací bezrozměrné. Pokud v některých tabulkách chybí u clusteru Hydra hodnoty pro určitý počet procesorů, znamená to, že se úlohu nepodařilo korektně dokončit. Ze stejného důvodu v některých případech chybí dokonce i celé tabulky.

## Krychle

| | 211014 elementů, původní nastavení předpodmiňovače | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | *proc.* | *1* | *2* | *4* | *6* | *8* | *10* | *12* | *16* | *20* | *24* | *32* |
| Hydra | *WHOLE PROG* | 160,4 | 133,3 | 229,9 | 556,3 | 645,1 | 107,5 | 1721,1 | 517,7 | | | |
| | | 1,00 | 0,60 | 0,17 | 0,05 | 0,03 | 0,15 | 0,01 | 0,02 | | | |
| | *SOLVING MH SYS* | 21,5 | 18,6 | 126,7 | 455,8 | 545,6 | 10,5 | 1623,3 | 417,5 | | | |
| | | 1,00 | 0,58 | 0,04 | 0,01 | 0,00 | 0,21 | 0,00 | 0,00 | | | |
| | *iterace* | 53 | 58 | 1785 | 10000 | 10000 | 53 | 10000 | 10000 | | | |
| | *efekt. 1. iterace* | 1,00 | 0,63 | 1,43 | 1,48 | 0,93 | 0,21 | 0,21 | 0,61 | | | |
| Rex | *WHOLE PROG* | 71,9 | 57,0 | 39,6 | 173,2 | 474,9 | 29,8 | 28,2 | 135,2 | 222,5 | 39,3 | 153,3 |
| | *- ef.* | 1,00 | 0,63 | 0,45 | 0,07 | 0,02 | 0,24 | 0,21 | 0,03 | 0,02 | 0,08 | 0,01 |
| | *SOLVING MH SYS* | 27,7 | 22,8 | 12,5 | 147,1 | 450,1 | 5,8 | 5,2 | 112,5 | 199,5 | 16,5 | 130,6 |
| | *- ef.* | 1,00 | 0,61 | 0,55 | 0,03 | 0,01 | 0,48 | 0,44 | 0,02 | 0,01 | 0,07 | 0,01 |
| | *iterace* | 53 | 53 | 54 | 2542 | 10000 | 54 | 55 | 4338 | 10000 | 826 | 10000 |
| | *efekt. 1 iterace* | 1,00 | 0,61 | 0,56 | 1,50 | 1,45 | 0,49 | 0,46 | 1,26 | 1,31 | 1,09 | 1,25 |

| | 211014 elementů, upravené nastavení předpodmiňovače | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | *proc.* | *1* | *2* | *4* | *6* | *8* | *10* | *12* | *16* | *20* | *24* | *32* |
| Rex | *WHOLE PROG* | 100,4 | 64,2 | 41,7 | 34,2 | 32,3 | 30,1 | 28,8 | 27,4 | 25,9 | 25,4 | 24,6 |
| | *- ef.* | 1,00 | 0,78 | 0,6 | 0,49 | 0,39 | 0,33 | 0,29 | 0,23 | 0,19 | 0,16 | 0,13 |
| | *SOLVING MH SYS* | 59,4 | 33,6 | 17,3 | 11,7 | 9,8 | 8,3 | 7,6 | 5,6 | 4,4 | 3,8 | 3,2 |
| | *- ef.* | 1,00 | 0,88 | 0,86 | 0,85 | 0,76 | 0,72 | 0,66 | 0,66 | 0,68 | 0,65 | 0,57 |
| | *iterace* | 24 | 24 | 24 | 24 | 25 | 24 | 26 | 25 | 23 | 23 | 25 |

| | test škálovatelnosti | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | *proc.* | *1* | *2* | *4* | *6* | *8* | *10* | *12* |
| Rex | *velikost úlohy* | 22415 | 37543 | 76571 | 117874 | 159197 | 211014 | 267156 |
| | *WHOLE PROG* | 7,42 | 9,09 | 16,09 | 20,37 | 25,02 | 31,10 | 36,58 |
| | *SOLVING MH SYS* | 3,39 | 4,06 | 6,91 | 7,32 | 8,20 | 9,19 | 8,87 |
| | *iterace* | 11 | 12 | 17 | 20 | 22 | 24 | 25 |
| | *efektivita* | 1,00 | 0,70 | 0,56 | 0,49 | 0,42 | 0,39 | 0,38 |

## 2D proužek

| | velikost 60314 elementů | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | *proc.* | *1* | *2* | *4* | *6* | *8* | *10* | *12* | *16* | *20* | *24* | *32* |
| Hydra | *WHOLE PROG* | 46,73 | 33,60 | 26,42 | 24,01 | 23,20 | 22,80 | 22,01 | 21,67 | | | |
| | *- ef.* | 1,00 | 0,70 | 0,44 | 0,32 | 0,25 | 0,20 | 0,18 | 0,13 | | | |
| | *SOLVING MH SYS* | 3,65 | 2,12 | 1,63 | 1,44 | 1,24 | 1,22 | 1,03 | 1,02 | | | |
| | *- ef.* | 1,00 | 0,86 | 0,56 | 0,42 | 0,37 | 0,30 | 0,30 | 0,22 | | | |
| | *TRANSPORT* | 22,89 | 11,93 | 7,20 | 4,37 | 3,45 | 3,01 | 2,65 | 2,12 | | | |
| | *- ef.* | 1,00 | 0,96 | 0,79 | 0,87 | 0,83 | 0,76 | 0,72 | 0,67 | | | |
| Rex | *WHOLE PROG* | 47,78 | 34,64 | 26,86 | 23,82 | 22,88 | 21,94 | 22,06 | 21,46 | 20,76 | 20,72 | 21,60 |
| | *- ef.* | 1,00 | 0,69 | 0,44 | 0,33 | 0,26 | 0,22 | 0,18 | 0,14 | 0,12 | 0,10 | 0,07 |
| | *SOLVING MH SYS* | 4,04 | 3,80 | 2,06 | 1,46 | 1,24 | 1,04 | 0,98 | 0,84 | 0,80 | 0,74 | 0,70 |
| | *- ef.* | 1,00 | 0,53 | 0,49 | 0,46 | 0,41 | 0,39 | 0,34 | 0,30 | 0,25 | 0,23 | 0,18 |
| | *TRANSPORT* | 25,30 | 12,64 | 6,48 | 4,38 | 3,42 | 2,78 | 2,42 | 1,96 | 1,64 | 1,44 | 1,28 |
| | *- ef.* | 1,00 | 1,00 | 0,98 | 0,96 | 0,92 | 0,91 | 0,87 | 0,81 | 0,77 | 0,73 | 0,62 |

## 3D proužek

| | velikost 74654 elementů | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | *proc.* | *1* | *2* | *4* | *6* | *8* | *10* | *12* | *16* | *20* | *24* | *32* |
| Hydra | *WHOLE PROG* | 98,43 | 85,87 | 69,08 | 63,40 | 60,60 | 57,30 | | | | | |
| | *- ef.* | 1,00 | 0,57 | 0,36 | 0,26 | 0,20 | 0,17 | | | | | |
| | *SOLVING MH SYS* | 8,04 | 6,51 | 3,40 | 3,17 | 2,66 | 2,34 | | | | | |
| | *- ef.* | 1,00 | 0,62 | 0,59 | 0,42 | 0,38 | 0,34 | | | | | |
| | *TRANSPORT* | 17,23 | 9,12 | 5,20 | 3,68 | 3,12 | 2,78 | | | | | |
| | *- ef.* | 1,00 | 0,94 | 0,83 | 0,78 | 0,69 | 0,62 | | | | | |
| Rex | *WHOLE PROG* | 61,24 | 48,46 | 39,42 | 35,58 | 34,2 | 34,3 | 33,54 | 32,58 | 32,22 | 31,98 | 32,06 |
| | *- ef.* | 1,00 | 0,63 | 0,39 | 0,29 | 0,22 | 0,18 | 0,15 | 0,12 | 0,10 | 0,08 | 0,06 |
| | *SOLVING MH SYS* | 10,92 | 8,40 | 4,38 | 2,88 | 2,24 | 1,98 | 1,72 | 1,44 | 1,28 | 1,18 | 1,06 |
| | *- ef.* | 1,00 | 0,65 | 0,62 | 0,63 | 0,61 | 0,55 | 0,53 | 0,47 | 0,43 | 0,39 | 0,32 |
| | *TRANSPORT* | 19,96 | 10,06 | 5,40 | 3,56 | 2,78 | 2,36 | 1,96 | 1,58 | 1,30 | 1,22 | 0,98 |
| | *- ef.* | 1,00 | 0,99 | 0,92 | 0,93 | 0,90 | 0,85 | 0,85 | 0,79 | 0,77 | 0,68 | 0,64 |

## Čtverec

| | velikost 92784 elementů, původní nastavení předpodmiňovače | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | *proc.* | *1* | *2* | *4* | *6* | *8* | *10* | *12* | *16* | *20* | *24* | *32* |
| Hydra | *WHOLE PROG* | 56,79 | 67,85 | 63,18 | 61,44 | 62,31 | 66,33 | 62,50 | 68,29 | | | |
| | *- ef.* | 1,00 | 0,42 | 0,22 | 0,15 | 0,11 | 0,09 | 0,08 | 0,05 | | | |
| | *SOLVING MH SYS* | 5,20 | 4,34 | 3,87 | 2,18 | 4,05 | 4,32 | 1,37 | 1,37 | | | |
| | *- ef.* | 1,00 | 0,60 | 0,34 | 0,40 | 0,16 | 0,12 | 0,32 | 0,24 | | | |
| | *TRANSPORT* | 23,07 | 12,92 | 6,56 | 4,83 | 3,67 | 3,04 | 2,65 | 2,21 | | | |
| | *- ef.* | 1,00 | 0,89 | 0,88 | 0,80 | 0,79 | 0,76 | 0,73 | 0,65 | | | |
| | *iterace* | 149 | 181 | 218 | 175 | 340 | 350 | 185 | 268 | | | |
| Rex | *WHOLE PROG* | 48,64 | 33,15 | 25,99 | 20,89 | 20,37 | 20,03 | 17,81 | 17,60 | 17,77 | 30,39 | 16,36 |
| | *- ef.* | 1,00 | 0,73 | 0,47 | 0,39 | 0,30 | 0,24 | 0,23 | 0,17 | 0,14 | 0,07 | 0,09 |
| | *SOLVING MH SYS* | 7,72 | 5,68 | 4,98 | 1,90 | 2,50 | 3,07 | 1,12 | 1,23 | 0,97 | 14,49 | 0,75 |
| | *- ef.* | 1,00 | 0,68 | 0,39 | 0,68 | 0,39 | 0,25 | 0,57 | 0,39 | 0,40 | 0,02 | 0,32 |
| | *TRANSPORT* | 26,09 | 13,24 | 6,71 | 4,92 | 3,60 | 2,90 | 2,71 | 1,92 | 1,69 | 1,38 | 1,15 |
| | *- ef.* | 1,00 | 0,99 | 0,97 | 0,88 | 0,91 | 0,90 | 0,80 | 0,85 | 0,77 | 0,79 | 0,71 |
| | *iterace* | 150 | 170 | 353 | 149 | 416 | 764 | 173 | 329 | 258 | 10000 | 241 |

| | velikost 92784 elementů, upravené nastavení předpodmiňovače | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | *proc.* | *1* | *2* | *4* | *6* | *8* | *10* | *12* | *16* | *20* | *24* | *32* |
| Hydra | *WHOLE PROG* | 84,63 | 64,51 | 50,77 | 37,57 | 31,15 | 28,92 | | | | | |
| | *- ef.* | 1,00 | 0,66 | 0,42 | 0,38 | 0,34 | 0,29 | | | | | |
| | *SOLVING MH SYS* | 6,53 | 5,03 | 3,50 | 2,38 | 1,82 | 1,48 | | | | | |
| | *- ef.* | 1,00 | 0,65 | 0,47 | 0,46 | 0,45 | 0,44 | | | | | |
| | *TRANSPORT* | 21,71 | 12,46 | 6,55 | 4,45 | 3,63 | 3,02 | | | | | |
| | *- ef.* | 1,00 | 0,87 | 0,83 | 0,81 | 0,75 | 0,72 | | | | | |
| | *iterace* | 74 | 74 | 76 | 73 | 73 | 71 | | | | | |
| Rex | *WHOLE PROG* | 50,60 | 33,53 | 24,50 | 20,68 | 19,14 | 19,04 | 18,10 | 17,24 | 16,82 | 16,49 | 16,32 |
| | *- ef.* | 1,00 | 0,75 | 0,52 | 0,41 | 0,33 | 0,27 | 0,23 | 0,18 | 0,15 | 0,13 | 0,10 |
| | *SOLVING MH SYS* | 10,43 | 6,10 | 3,49 | 2,09 | 1,68 | 1,31 | 1,13 | 0,93 | 0,82 | 0,73 | 0,64 |
| | *- ef.* | 1,00 | 0,85 | 0,75 | 0,83 | 0,78 | 0,80 | 0,77 | 0,70 | 0,64 | 0,60 | 0,51 |
| | *TRANSPORT* | 25,96 | 13,23 | 6,77 | 4,79 | 3,49 | 3,24 | 2,47 | 1,91 | 1,60 | 1,40 | 1,24 |
| | *- ef.* | 1,00 | 0,98 | 0,96 | 0,90 | 0,93 | 0,80 | 0,88 | 0,85 | 0,81 | 0,77 | 0,65 |
| | *iterace* | 75 | 69 | 74 | 72 | 75 | 69 | 72 | 72 | 75 | 69 | 66 |

## Melechov

| | velikost 111563 elementů | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | *proc* | *1* | *2* | *4* | *6* | *8* | *10* | *12* | *16* | *20* | *24* | *32* |
| Hydra | *WHOLE PROG* | 712,34 | 399,75 | 287,35 | 198,01 | 157,69 | 173,03 | 149,87 | 145,5 | | | |
| | *- ef.* | 1,00 | 0,89 | 0,62 | 0,60 | 0,56 | 0,41 | 0,40 | 0,31 | | | |
| | *SOLVING MH SYS* | 5,59 | 4,88 | 3,29 | 2,99 | 2,62 | 4,70 | 4,62 | 5,23 | | | |
| | *- ef.* | 1,00 | 0,57 | 0,42 | 0,31 | 0,27 | 0,12 | 0,10 | 0,07 | | | |
| | *TRANSPORT* | 617,63 | 321,91 | 218,98 | 133,25 | 94,02 | 106,64 | 84,94 | 80,31 | | | |
| | *- ef.* | 1,00 | 0,96 | 0,71 | 0,77 | 0,82 | 0,58 | 0,61 | 0,48 | | | |
| Rex | *WHOLE PROG* | 678,65 | 360,15 | 203,89 | 147,41 | 122,35 | 114,04 | 101,46 | 87,53 | 88,65 | 81,54 | 77,89 |
| | *- ef.* | 1,00 | 0,94 | 0,83 | 0,77 | 0,69 | 0,60 | 0,56 | 0,48 | 0,38 | 0,35 | 0,27 |
| | *SOLVING MH SYS* | 8,01 | 7,17 | 4,28 | 3,10 | 2,62 | 2,91 | 1,94 | 1,62 | 1,43 | 1,20 | 1,10 |
| | *- ef.* | 1,00 | 0,56 | 0,47 | 0,43 | 0,38 | 0,28 | 0,34 | 0,31 | 0,28 | 0,28 | 0,23 |
| | *TRANSPORT* | 637,44 | 329,91 | 182,28 | 128,82 | 105,26 | 97,34 | 86,14 | 72,19 | 63,58 | 57,80 | 51,33 |
| | *- ef.* | 1,00 | 0,97 | 0,87 | 0,82 | 0,76 | 0,65 | 0,62 | 0,55 | 0,50 | 0,46 | 0,39 |

## Decovalex

| | velikost 60084 elementů; nulový tlak | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | *proc* | *1* | *2* | *4* | *6* | *8* | *10* | *12* | *16* | *20* | *24* | *32* |
| Rex | *WHOLE PROG* | 32,42 | 17,69 | 9,55 | 7,43 | 6,75 | 6,19 | 5,69 | 5,57 | 5,61 | 5,22 | 5,39 |
| | *- ef.* | 1,00 | 0,92 | 0,85 | 0,73 | 0,60 | 0,52 | 0,47 | 0,36 | 0,29 | 0,26 | 0,19 |
| | *SOLVING MH SYS* | 1,95 | 1,32 | 0,70 | 0,54 | 0,44 | 0,40 | 0,35 | 0,31 | 0,29 | 0,28 | 0,27 |
| | *- ef.* | 1,00 | 0,74 | 0,70 | 0,60 | 0,55 | 0,49 | 0,46 | 0,39 | 0,34 | 0,29 | 0,23 |
| | *TRANSPORT* | 26,15 | 12,34 | 4,77 | 2,82 | 2,08 | 1,54 | 1,32 | 1,02 | 0,98 | 0,83 | 0,75 |
| | *- ef.* | 1,00 | 1,06 | 1,37 | 1,55 | 1,57 | 1,70 | 1,65 | 1,60 | 1,33 | 1,31 | 1,09 |